Java 8 to 21

Explore and work with the cutting-edge features of Java 21

Shai Almog

bpb

Java 8 to 21

Explore and work with the cutting-edge features of Java 21

Shai Almog

www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-922

www.bpbonline.com

Dedicated to

Tao & Tara

About the Author

Shai is an author, entrepreneur, blogger, open-source hacker, speaker, Java rockstar, developer advocate and more. He is a former Sun/Oracle engineer with 30+ years of professional experience. Shai built **Java Virtual Machines (JVMs)**, development tools, mobile phone environments, banking systems, startup/enterprise backends, user interfaces, development frameworks and much more.

Shai is on the advisory board for multiple organizations including dzone, dev network, and so on. Shai speaks at conferences all over the world and shared the stage with luminaries such as James Gosling (father of Java).

About the Reviewer

Ravi Rajpurohit is a highly accomplished Software Engineer, currently holding the position of Manager at Morgan Stanley in Bangalore. With a career spanning various prestigious organizations, Ravi has amassed a wealth of experience and expertise in the field. His journey began as an intern at ISRO RRSC-W in Jodhpur, where he laid the foundation for his career in software engineering. He further solidified his skills while working as a Software Engineer at Confluxsys Private Limited in Pune. Ravi's talent and dedication led him to excel as a Senior Software Engineer at Publicis Sapient in Bangalore, where he contributed to the development of cutting-edge software solutions. Proficient in Python, Java, PHP, and other programming languages, Ravi possesses a versatile skill set that allows him to tackle complex technical challenges. Outside of his professional pursuits, Ravi finds joy in sharing his knowledge through technical teaching during his spare time. Originally from Jodhpur, Ravi brings a unique perspective and strong work ethic to his endeavors, making him a valuable asset in the software engineering realm.

Preface

Java has a big problem. It is often perceived as old, but Python is older, and JavaScript is its contemporary. Why is that?

Java has had continuous popularity and success for nearly three decades. That means that most of us worked with Java 1.4, which was released two decades ago. Unlike any other platform out there, Java is still compatible with that release. That is fantastic but also creates a sense of disconnect. Developers compare that highly outdated version of Java to modern incarnation of other languages or platforms. That is an unfair comparison, and Java is a victim of its own success.

There are several sources for information about new Java features but there is a lack in a comprehensive introductory guide to modern Java, that carries us from that old version to the modern world. With this book, we hope to fill that gap, to answer the question of "is this still the right way to solve this problem?".

This book is divided into 10 chapters. They cover Java basics, OOP basics and continue to cover everything we need to know, in order to build a modern Java backend. Here is a brief description for each chapter:

Chapter 1: Hello Java – In this chapter, we will review the basics of the language and tooling, to make sure we are all on the same page. We focus on Java 8 syntax and as we move forward in the book, we will discuss the newer features of Java, all the way to version 21.

Chapter 2: OOP Patterns – In this chapter, we explore the best ways to leverage OOP, what we would consider as good design and what type of OOP principles are expressed in the Java API.

Chapter 3: 8 to 21 to GraalVM – Java went through many pivotal releases. Java 1.1 introduced inner classes. Java 2 introduced Swing and collections. Java 5 introduced Generics. However, one of the most pivotal releases was Java 8, which introduced a slew of wide sweeping changes. Up until Java 8, the releases of Java versions were irregular although they tended to follow a two-year cadence. With the release of Java 9, this changed. We now have a 6-month release cycle. This pace made the period between Java 8 and Java 21 relatively short. We review the changes

between these releases in this chapter.

Chapter 4: Modern Threading – In this chapter, we will discuss the evolution of Java threading from the beginning, all the way to the future. We will explain the trade-offs and challenges and we will also discuss the best ways to write highly concurrent code moving forward.

Chapter 5: It's Springtime in Java – Spring brings together libraries and tools from multiple different projects, to create a unified experience. Spring is also highly configurable; it is not limited to backend or database driven applications.

Chapter 6: Testing and CI – In this chapter, we will cover some common tools in the world of testing such as JUnit and Mockito. We will cover GitHub Actions when discussing CI due to its wide availability and free quota. We will also discuss some high-level concepts such as **Test-Driven Development (TDD)**.

Chapter 7: Docker, Kubernetes and Spring Native – The original thought behind this chapter was around cloud native development. However, that acronym is somewhat vague. In the old days, we would take the resulting jar files we would build and place them on an application server connected to the internet. These days are long gone, containers and orchestration have completely changed the way applications are deployed in production.

Chapter 8: Microservices – The rise of Kubernetes has made the Microservices approach far more commonplace. The cost of spinning up and managing multiple smaller servers became manageable and as a result, drove a massive move in that direction. Microservices bring a great deal to the table, they are easier to build and often easier to scale.

Chapter 9: Serverless – The complexities of Monoliths and Microservices pushed some developers towards an easier route. Serverless is a problematic branding term indicating that we no longer need to manage our server, only the application. The serverless provider takes over the complexity of spinning up container instances for us as long as we follow some guidelines in application development.

Chapter 10: Monitoring and Observability – Monitoring and observability are essential aspects of software development that help developers and operations teams ensure the reliability, stability, and performance of their applications. As systems become increasingly complex, the need for effective monitoring and observability tools grows in tandem, providing a comprehensive understanding of the behavior and health of applications during development, deployment, and production. The Java ecosystem is no exception, offering a rich set of libraries and

tools designed to facilitate these critical tasks. This chapter will delve into the concepts of monitoring and observability in the context of Java applications, exploring the underlying principles, methodologies, and best practices that will enable developers to build robust, scalable, and resilient systems.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the Coloured Images of the book:

https://rebrand.ly/boau4e5

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Java-8-to-21**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and

eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Hello Java

- Introduction
- Structure
- Objectives
- Requirements
- Setting up a project
- Hello Java
- Principals of OOP
  - *Encapsulation*
  - *Inheritance*
  - *Polymorphism*
- Built-in types
  - *Arrays*
  - *Generics and Erasure*
- Debugging
  - *Debugging Java*
- Conclusion
- Points to remember
- Multiple choice questions
  - *Answers*

2. OOP Patterns

- Introduction
- Structure
- Objectives
- Why OOP?
- Basic Object-Oriented Design
- Common patterns
  - *Singleton*
  - *Factory*
  - *Builder*

Adapter
Façade
Proxy
Observer
Command
Iterator
Immutability
Functional programming
Conclusion
Points to remember
Multiple choice questions
Answers

3. 8 to 21 to GraalVM
Introduction
Structure
Objectives
Java 8—the baseline
Lambda expressions
Method references
Default methods
Streams
Annotation changes
Method parameters reflection
New date and time APIs
VM changes
Modules AKA Jigsaw (Java 9)
Shenandoah (Java 12)
Not your fathers stop the world mark sweep
Generational garbage collection
Concurrent versus parallel garbage collector
Serial collector
Parallel collector or throughput collector
G1 Garbage Collector
Z Garbage Collector (ZGC)
Shenandoah
Microbenchmark Suite (Java 12)
Virtual Threads—Loom (Java 21)
Loom
Deprecation of finalization (Java 9)

UTF-8 by default (Java 18)
Language changes
Try with resources
Private methods in interfaces (Java 9)
var keyword (Java 10)
Switch expression (Java 14)
Sealed classes (Java 17)
Pattern-matching instanceof (Java 16)
Text blocks (Java 15)
Records (Java 16)
Record patterns (Java 19 preview)
String templates (Java 21 preview)
Unnamed patterns and variables (Java 21 preview)
APIs
HttpClient (Java 11)
Foreign function and memory API—Panama (Java 19)
Structured concurrency (Java 19)
Serialization filtering (Java 9)
Scoped values (Java 20)
Sequenced collection (Java 21)
Future
GraalVM
Valhalla
Conclusion
Points to remember
Multiple choice questions
Answers

4. Modern Threading
Introduction
Structure
Objectives
History
Concepts
Thread safety
Mutex (Lock)
Wait and notify (monitor)
Deadlock

Race conditions
Thread pools
Executors
Locks
Synchronizers
Atomic
Futures
Collections and queues
Conclusion
Points to remember
Multiple choice questions
Answers

5. It's Springtime in Java
Introduction
Structure
Objectives
History and origin
Inversion of Control, Dependency Injection, and Aspect Oriented Programming
Hello Spring Boot
A REST API
Spring MVC and Thymeleaf
SQL and JDBC
SQL basics
Java Database Connectivity
Java Persistence Architecture
Error handling
Conclusion
Points to remember
Multiple choice questions
Answers

6. Testing and CI
Introduction
Structure
Objectives
Testing theory
JUnit

Mockito
Performance matters
What should we mock?
Test Driven Development
The problem with TDD
Continuous Integration
Continuous Integration tools
Cloud versus on premise
Agent statefullness
GitHub Actions
Branch protection
Linting
Conclusion
Points to remember
Multiple choice questions
Answers

7. Docker, Kubernetes, and Spring Native
Introduction
Structure
Objectives
Containers
Docker
Orchestration
Kubernetes (k8s)
The easy way—Skaffold
Infrastructure as Code (IaC)
Spring Native
Getting started with Spring Native
Cloud Native
Alternatives to Spring Native
Conclusion
Points to remember
Multiple choice questions
Answers

8. Microservices
Introduction
Structure

Objectives
Microservices versus small Monoliths
Service mesh
Authentication and authorization
Eventual consistency
Messaging
RabbitMQ
Apache Kafka
Spring cloud stream
Publish subscribe and point to point
Monolith first
Modular Monolith
Modulith
Other modules
The benefit
Conclusion
Points to remember
Multiple choice questions
Answers

9. Serverless
Introduction
Structure
Objectives
What is serverless
Using AWS Lambda
GraalVM and Monolith serverless
The cloud ecosystem
Conclusion
Points to remember
Multiple choice questions
Answers

10. Monitoring and Observability
Introduction
Structure
Objectives
What is monitoring?
Pillars of observability

The three pillars of observability
What makes a system observable?
Prometheus
Grafana
Micrometer
Actuator
Enabling and configuring Spring Boot actuator
Exposing custom information via Spring Boot actuator
Developer observability
Injecting logs
Snapshots/captures and non-breaking breakpoints
PII reduction and blocklists
Conclusion
Points to remember
Multiple choice questions
Answers

Index

Chapter 1
Hello Java

Introduction

Java is one of the most ubiquitous programming languages on earth. One of its biggest selling points is its simplicity. In fact, the father of Java, *James Gosling*, used to describe his approach to the design of Java with the following metaphor.

When James had to move to a new residence, he would pack all his belongings into well-labeled boxes. Then, he would only unpack the stuff he needed. Six months later, he would throw away all the other boxes without looking at the content. If he looked at the content, he would probably keep it. Java was designed according to the principle of minimalism. Over the years, it grew, but it is still a small language with a massive API and ecosystem.

In this chapter, we will review the basics of the language and tooling to make sure we are all on the same page. We will focus on Java 8 syntax, and as we move forward in the book, we will discuss the newer features of Java, all the way to version 20.

Structure

In this chapter, we will discuss the following topics:

- Requirements
- Setting up a project
- Hello Java
- Principals of OOP
    - Encapsulation
    - Inheritance

    - Polymorphism
- Built-in types
    - Arrays
    - Generics and Erasure
- Debugging
    - Debugging Java

Objectives

By the end of this chapter, the reader will learn the basics of working with Java. This book assumes some basic prior familiarity with Java. Regardless we chose to approach this chapter as a blank slate since your familiarity might be significantly out of date. By the end of this chapter, you will be able to read simple Java code; as the book progresses, we will build on top of that knowledge. As such, this chapter is the foundation for the rest of the book.

Requirements

To get started, we need to install the Java JDK, and in our case, OpenJDK is recommended. OpenJDK is the open-source version of the Java virtual machine, which we need in order to run Java applications. There are commercial packages of the JDK as well as the Oracle official bundle. However, OpenJDK is free, portable, and supported indefinitely, so a variant of OpenJDK is highly recommended. There is a comprehensive tool to locate the OpenJDK distribution that fits your requirements on the foojay.io site[1]. You can check out SDKMAN[2] if using a Mac, Linux, or the Linux subsystem on Windows.

Other than that, you will need to install IntelliJ/IDEA[3], which is the leading Java IDE. The community edition is sufficient for most novices, but the Ultimate edition is recommended, especially for working with Spring and Web.

Setting up a project

Java is simple. This facilitated an enormous ecosystem around it and, as a result, added complexity to Java as a whole. We can compile a single Java source file using the **javac** command line. But this is not practical for significant applications. A real-world application is comprised of multiple files, with interaction among the various pieces, and is often packaged in a novel way. There are many complexities in the build process, including the management of dependencies. That is where build tools come in. At the time of this writing, there are three common build tools for Java as

follows:

- **Maven**: This is the tool we will discuss in this book. It is the market leader, although a bit older. It uses XML syntax by default and is somewhat clunky. However, it has a huge ecosystem and is very mature.
- **Gradle**: Uses a syntax based on Groovy or Kotlin for the build scripts. Works with Mavens dependency system, making the migration from Maven easy.
- **Ant**: A legacy system based on XML and a precursor to Maven. It is listed for it its completeness, as we still run into ant scripts occasionally.

All of these tools can be used effectively from the command line, but we do not need to do that. Java is particularly powerful when working within the confines of the IDE, where we can leverage its automation to instantly spot problems and investigate our code. To get started, we launch IntelliJ/IDEA and create a new project[4], as seen in *figures 1.1* and *1.2*:

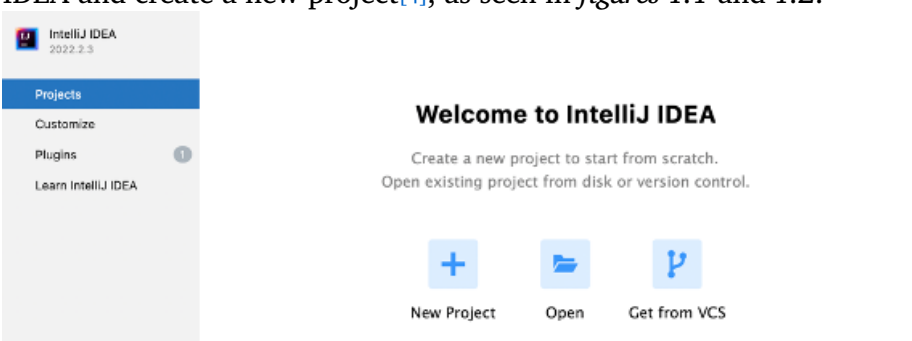

Figure 1.1: *IntelliJ/IDEA new project window*

Refer to *figure 1.2* to see the steps for the creation of a new project:

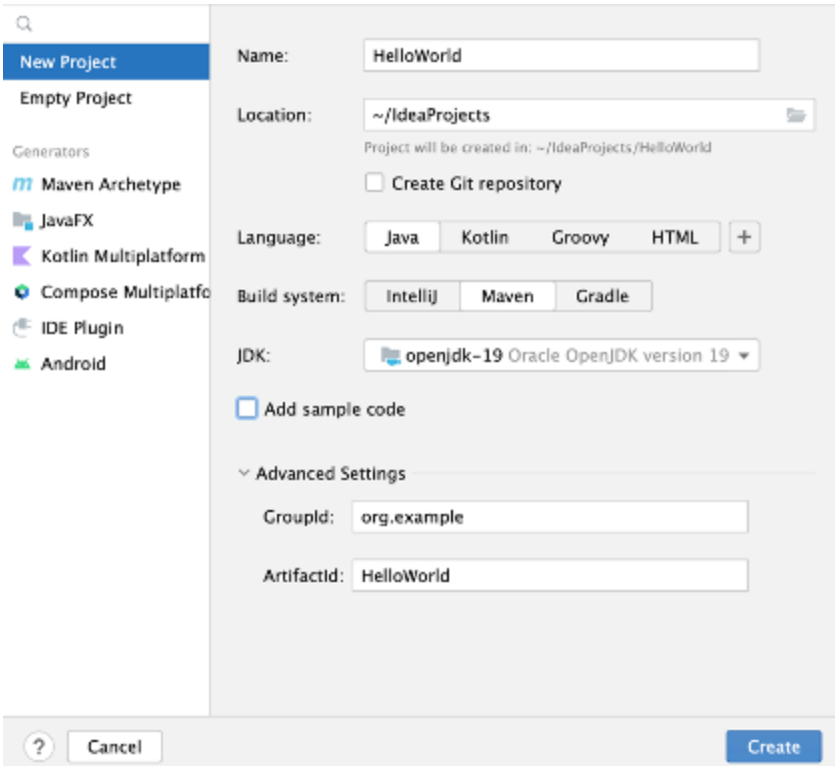

Figure 1.2: *IntelliJ/IDEA creating a new project*

When creating a new project, make sure to select **Maven** as the build system and a new version of the JDK. Notice that you can download a version of the JDK directly from the combo box to pick a JDK, as shown in *figure 1.3*.

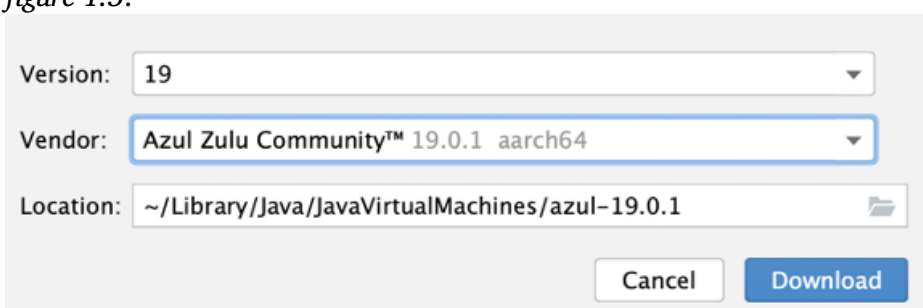

Figure 1.3: *Download and install a new JDK from within IntelliJ/IDEA*

Once we create a new project, we are faced with the UI shown in *figure 1.4*. We have expanded the tree on the left-hand side to show the Java directory. This is where our Java source code should be placed. On the right side, we can see the maven pom file, which is not important at this point. This gives us the environment where we can start writing code.

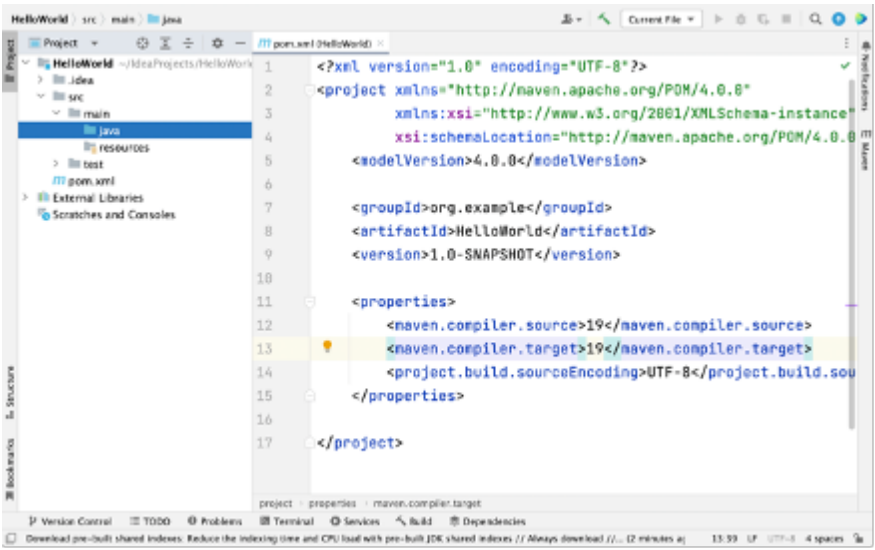

Figure 1.4: *New Maven project in IntelliJ/IDEA*

Hello Java

Let us start with the most basic "hello world" example we can make. Right-click the Java directory in the project, as shown in *figure 1.5,* and select **New | Java Class**. When prompted, we can enter the name of the new class and press enter. We went with **HelloWorld**. Names have a capital first letter by convention (but it is not required). They followed the convention of capitalizing every word. Names cannot include spaces and various other characters. They cannot start with a number either but can be Unicode characters from arbitrary languages (although this is uncommon). The name cannot be one of the reserved Java keywords[5].

Figure 1.5 features the New Java Class Context Action:

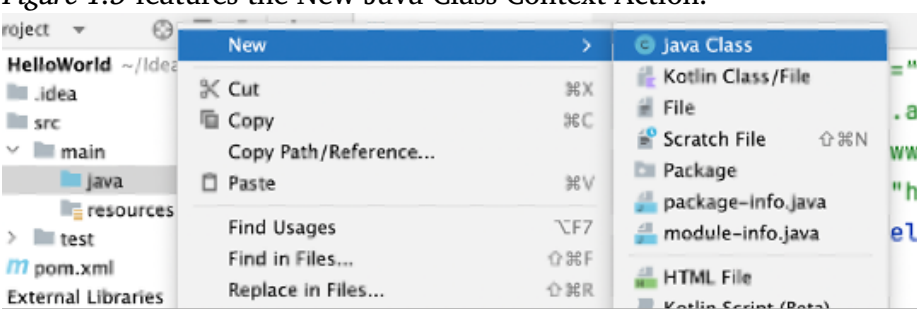

Figure 1.5: *New Java Class Context Action*

Figure 1.6 features the New Java Class Context Action:

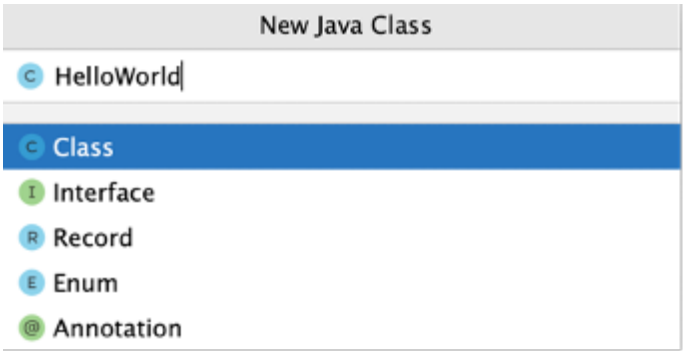

Figure 1.6: *New Java Class Prompt dialog*

This creates a file named **HelloWorld.java** in Java. In order for a class to be public, it must reside in a source file with the same name. Notice that since the language is case-sensitive, the public class **HelloWorld** cannot reside in the file **helloworld.java**. A public class is exposed to usage outside of its package. We will discuss packages soon enough. Now that we understand that let us create our first Java application.

```
1. public class HelloWorld {
2. public static void main(String[] argv) {
3. System.out.println("Hello World");
4. }
5. }
```

Let us go over the lines in the code one by one. In Line 1, we start with the **public** keyword. This indicates that we wish to export this class to external packages. In this case, since we want to run the class, it must be public. The next keyword is **class**, which is a basic building block of objects in Java. In Java, almost everything is part of a class in one way or another. Classes let us package code and data (methods and fields) together and work as a single cohesive element. This is a big subject that we will discuss in the following section.

In Line 2, we start with the public again. Elements within the class can have different visibility levels. Within the class, you have full access to everything, and visibility applies to other classes. If we remove the public keyword from the class, it can still be used by other classes in the same package but cannot be used out of the package. The same is true for the elements we write in the class. In this case, it is a method that is an operation we can perform. This method can be accessed by everyone because it is public. Methods (and fields) can have the following visibility levels:

- **public**: Full access by anyone with access to the class. Notice that if

the class is not public and the method is public, it would still not be visible to everyone. Only those who can access the class.

- **protected**: This is like the default access but also allows access to subclasses, even if they are outside of the current package. We will discuss subclassing soon.
- **[default]**: Unlike the others, this is not a keyword. This is the default mode when we do not specify visibility and just leave out one of the other keywords. The default visibility is package private. That means only classes within the package have access to the method or field.
- **private**: This is the strictest visibility level. Elements marked as private are only visible from within the class.

The next keyword is **void**, which means that the method does not return a value. After that, we have the name of the method, which is "main" and the arguments passed to this method. The arguments are an array of strings named **argv**.

The body of the method references the **System** class, which has a field named **out**. We invoke the **println** public method on that object and pass the string **Hello World** as the argument. This prints **Hello World**, as can be seen in *figure 1.7*.

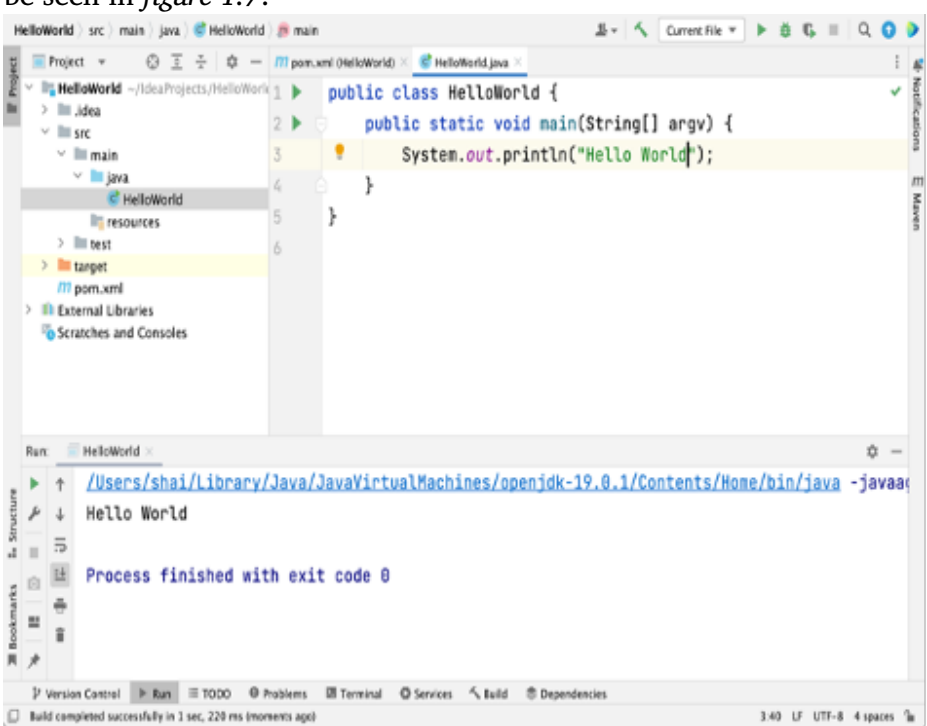

Figure 1.7: *After pressing the green play button on the top, the application runs; we can see the output in the Console section at the bottom of the screenshot*

Principals of OOP

Let us pause for a second and take a step back. Why is it important to place methods in classes? Why are we making such a big deal about visibility attributes such as the public? This all fits into the three principles of **Object Oriented Programming** (**OOP**).

Encapsulation is the first and arguably most important principle of the three. It means that the data of the object and the operations on that data are packaged together in one class. But it has another important aspect: hiding. When we encapsulate data, we hide implementation details. We will start with a simple example.

Encapsulation

Let us consider, for example, that your daughter is learning simple fractions at school. You want to help her practice that. You want to create an application that will help her practice simple fractions. In a simple fraction, we have two numbers: a numerator and a denominator. The numerator is the number on top of the fraction, and the denominator is the number on the bottom[6]. We can represent a fraction like the following:

```
1. public class Fraction {
2. public int numerator;
3. public int denominator;
4. }
```

This code does not include any encapsulation whatsoever, and we left both fields public. Let us see how we can use it to implement a simplistic math equation. Notice that we are taking a very simple approach here because this is a demo. We will package the logic into a method so that we can add two fractions easily:

```
1. public static Fraction addFractions(Fraction first, Fraction second) {
2. var newFraction = new Fraction();
3. newFraction.numerator =
4. first.numerator * second.denominator +
5. second.numerator * first.denominator;
6. newFraction.denominator =
7. first.denominator * second.denominator;
8. return newFraction;
9. }
```

The code is relatively simple, albeit a bit verbose. We access the fields and multiply them, add them, and assign the result to a newly created object.

We can now make simple usage of this API:

```
1. var first = new Fraction();
2. first.numerator = 1;
3. first.denominator = 2;
4.
5. var second = new Fraction();
6. second.numerator = 2;
7. second.denominator = 3;
8.
9. var result = addFractions(first, second);
10. System.out.println(
11. first.numerator + "/" + first.denominator + " + " +
12. second.numerator + "/" + second.denominator + " = " +
13. result.numerator + "/" + result.denominator);
```

The code is very simple; we create two objects. Assign the values representing 1/2 and 2/3, respectively. We then invoke the **addFractions** method that we defined before. Finally, we print the full equation. This is a bit verbose, but ultimately works. It can be made more efficient with additional methods, but it has some failings that cannot be fixed. Let us continue with the code.

```
1. var third = new Fraction();
2. third.numerator = 1;
3. third.denominator = 2;
4.
5. var forth = new Fraction();
6. forth.numerator = 2;
7.
8. // bug forgot to change that to forth...
9. second.denominator = 3;
10.
11. var secondResult = addFractions(third, forth);
12. System.out.println(
13. third.numerator + "/" + third.denominator + " + " +
14. forth.numerator + "/" + forth.denominator + " = " +
15. secondResult.numerator + "/" + secondResult.denominator);
```

The fact that this code is duplicated and verbose is a problem. But the bigger problem is the 9th line. It assigns the value to the wrong variable,

resulting in a division by zero. If we run the code, we see the following:

1/2 + 2/3 = 7/6

1/2 + 2/0 = 4/0

Notice that the second line is wrong because the code was not meant to deal with division by zero. Since there is no encapsulation, we could not catch the illegal value of the field before usage. Let us take a second stab at this with encapsulation:

```
1. public class Fraction {
2. private final int numerator;
3. private final int denominator;
4.
5. public Fraction(int numerator, int denominator) {
6. this.numerator = numerator;
7. this.denominator = denominator;
8. if(denominator <= 0) {
9. throw new IllegalArgumentException("Invalid denominator: " +
   denominator);
10. }
11. }
12.
13. public Fraction add(Fraction other) {
14. int numerator = this.numerator * other.denominator +
15. other.numerator * denominator;
16. int denominator = this.denominator * other.denominator;
17. return new Fraction(numerator, denominator);
18. }
19.
20. @Override
21. public String toString() {
22. return numerator + "/" + denominator;
23. }
24. }
```

Let us review specific lines of code to understand what is different about this version of the class. In Lines 2 and 3, we define the same variables. They are private, which means that they are fully encapsulated and can only be accessed from within the same class. They are both final. That means they cannot be modified after assignment; they must be assigned in the constructor at the latest. This effectively makes this class immutable;

its content cannot be modified. Immutability is an important design principle as it promotes safer, more reliable code.

In Line 5, we define a constructor for the class that initializes both variables. Notice that we use the same name for the constructor arguments as the fields. This is completely optional but is a very common convention in Java. To distinguish between the arguments and the class fields, we prefix the fields with the keyword **this**. In Line 8, we explicitly throw an exception if the denominator is illegal. This prevents users from creating invalid objects intentionally or accidentally.

The add method on Line 13 includes many encapsulation benefits. It is no longer static and can be named **add** instead of **addFractions** since it is now directly associated with a fraction. It no longer needs a second argument since it uses the fields of this class.

Line 21 overrides the **toString** method of Java **Object**. This brings us to an inheritance which is the second principle of OOP. All objects in Java inherit from a class called **Object**, which defines a few important methods, including **toString**. This means that when we try to print the object, it will appear correctly. Notice that Line 20 includes the **@Override** annotation.

Annotations let us declare things about elements in the Java code; in this case, we indicate that we are replacing a method that is defined in the base class (**Object**), but we do not need to do that. It will work fine without the override annotation. The reason it is recommended to add that annotation is that if the method in the base class is removed or missing, we will get a compiler error. We will discuss inheritance in more detail soon enough. Let us look at the usage of this new class:

```
1. var first = new Fraction(1, 2);
2. var second = new Fraction(2, 3);
3. var result = first.add(second);
4. System.out.println(first + " + " + second + " = " + result);
5.
6. var third = new Fraction(1, 2);
7.
8. // will throw an exception...
9. var forth = new Fraction(2, 0);
10. var secondResult = third.add(forth);
11. System.out.println(third + " + " + forth + " = " + secondResult);
```

This is the full usage, including the “buggy” second block. Notice how much more concise it is. Line 3 is particularly satisfying in its simplicity. Notice that Lines 4 and 11 become trivial compared to the previous code.

Since **toString()** is built into Java, the code is the equivalent of writing:

```
1. first.toString() + " + " + second.toString() + " = " + result.toString()
```

When we run this version, the bug in Line 9 becomes even more obvious as we get a clear exception:

1/2 + 2/3 = 7/6

Exception in thread "main" java.lang.IllegalArgumentException: Invalid denominator: 0

at com.debugagent.ch01.encapsulation.Fraction.<init>(Fraction.java:

at com.debugagent.ch01.encapsulation.SampleUsage.main(SampleUsag

Notice that the exception points us to the file where the error occurred, that is, the class name and the line number. This makes it very easy to locate the code that triggered the problem and make a fix.

Java 14 introduced a new concept: **Records**. A Java record is a final class that has final fields. It is immutable. This seems like the ideal option for our fractions. Let us port our code to use records:

```
1. public record Fraction(int numerator, int denominator) {
2. public Fraction add(Fraction other) {
3. int numerator = this.numerator * other.denominator +
4. other.numerator * this.denominator;
5. int denominator = this.denominator * other.denominator;
6. return new Fraction(numerator, denominator);
7. }
8. }
```

This is the record equivalent of our fraction class or at least a close approximation. The usage code is identical if we use a record. However, there are two things missing here. The **toString()** method and the verification code. If we run this, you will see the following output:

Fraction[numerator=1, denominator=2] + Fraction[numerator=2, denominator=3] = Fraction[numerator=7, denominator=6]

Fraction[numerator=1, denominator=2] + Fraction[numerator=2, denominator=0] = Fraction[numerator=4, denominator=0]

This is due to the default implementation of **toString()** in records and the fact that we did not explicitly create a constructor. We can solve both

problems by creating a more verbose record:

```
1. public record Fraction(int numerator, int denominator) {
2. public Fraction(int numerator, int denominator) {
3. this.numerator = numerator;
4. this.denominator = denominator;
5. if(denominator <= 0) {
6. throw new IllegalArgumentException("Invalid denominator: " +
   denominator);
7. }
8. }
9.
10. public Fraction add(Fraction other) {
11. int numerator = this.numerator * other.denominator +
12. other.numerator * this.denominator;
13. int denominator = this.denominator * other.denominator;
14. return new Fraction(numerator, denominator);
15. }
16.
17. public String toString() {
18. return numerator + "/" + denominator;
19. }
20. }
```

It is still slightly smaller than the class and replaces the default implementations of the constructor and **toString()**. It is still worth it since it implements other methods, specifically **equals()** and **hashcode()**.

One final subject we should cover is packages. In the sample code for this chapter, you will find all the samples shown to you. They are all in a single project file, and all have the same names. This might seem odd. How can the **Fraction** class avoid collision with the **Fraction** record?

The answer is that they reside in different packages. The best practice in Java is to place all classes within packages representing their roles. The name of the package uses a reverse domain notation, followed by the name of the package. In a similar way to classes residing in files bearing the same name, we expect packages to reside in directories matching the package name. For example, in the following package:

```
1. package com.debugagent.ch01.records;
```

The IDE created a directory hierarchy matching **com/debugagent/ch01/**

records under the Java directory. Notice the name of the package. The author of the book owns the domain debugagent.com. By using the name that one owns in reverse, we make sure it will not collide with code that another developer might write. The following parts of the package name are up to you to decide. There is another abstraction of modules that we will discuss later.

Inheritance

The second principle of OOP is inheritance. We discussed it briefly in the encapsulation section but let us take a step back and discuss the basics both in OOP and in Java. Inheritance lets us base a new class on an existing one, where we can expose common functionality. Java includes the following two types of inheritance:

1. Implementation
2. Interface

With implementation inheritance, the subclass includes the state of the base class, as we can see in the following code:

```
1. public class Base {
2. protected int var;
3. }
4.
5. public Subclass extends Base {
6. public int getVar() {
7. return var;
8. }
9. }
```

A class in Java can only inherit from one class. But it can implement as many interfaces as it desires. An interface has many restrictions; it cannot declare fields and historically could not implement methods. It can do that starting with Java 8, but it is a limited feature since fields cannot be declared. Every field declared in the interface is immediately made **static**, **final**, and **public**. That makes all fields implicitly constant. All methods are implicitly public. In the following code, we implement an interface and are required by the compiler to implement the method defined in the interface.

```
1. public interface BaseInterface {
2. public void method();
3. }
4.
```

```
public Subclass implements BaseInterface {
public void method() {
// code...
}
}
```

As mentioned before, all Java objects derive from a single class called **Object**. This is implicit and cannot be disabled. If you inherit from a different class, then the root of the inheritance hierarchy will always be **Object**. Object defines several methods of interest mentioned before, but its most important method is **getClass()**. Every object in Java has a single **Class** object representing it. Notice that **class** with a lowercase c is a reserved word in Java with which we use to declare a class. **Class** with an uppercase C is a specific type in Java that we can use to get information about the current class.

One of the best tools in Java is JShell. It was introduced in Java 9 and provided a much-needed interactive shell for Java. Scripting languages often feature a **Read Eval Print Loop** (**REPL**). This is a command line tool we can use to experiment with the language dynamically. We can start JShell from the JDK command line. Let us use it to understand how the **Class** object works:

```
$ jshell
jshell> var myObject = new Object();
myObject ==> java.lang.Object@506e1b77

jshell> System.out.println(myObject.getClass().getName());
java.lang.Object

jshell>
System.out.println(Arrays.toString(myObject.getClass().getMethods()));
[public boolean java.lang.Object.equals(java.lang.Object), public
java.lang.String java.lang.Object.toString(), public native int
java.lang.Object.hashCode(), public final native java.lang.Class
java.lang.Object.getClass(), public final native void java.lang.Object.notify(),
public final native void java.lang.Object.notifyAll(), public final void
java.lang.Object.wait(long) throws java.lang.InterruptedException, public
final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException, public final void java.lang.Object.wait()
throws java.lang.InterruptedException]

jshell> /exit
| Goodbye
```

Let us go over what we have here. In Line 1, we launch the JShell tool from the command line. For this to work, the bin directory of the JDK needs to be in your system path. The lines starting with **jshell>** are the commands we sent. The following lines are responses from the REPL. We start by declaring a new variable called **myObject** in Line 2. Notice that the REPL prints out information after some operations, as we see in Line 3. It shows the assignment to a new variable.

In Line 5, we print out the name of the class that we get from the **Class** object. This is a very common scenario in Java. At Line 8, we print out all the methods in the class. Notice that the method **getMethods()**returns an array of methods. Arrays are objects in Java, but they do not override the **toString()** method. If we had printed that object, it would look similar to the output on Line 3. The Arrays class contains common utility methods for arrays, including a decent **toString()** method.

In Line 11, we can see the way to quit JShell. This is a fantastic tool, and we will use it to demonstrate some other great features in Java throughout the book.

Polymorphism

We glossed over some features of inheritance as they do not make as much sense without understanding polymorphism. Let us look at the following example:

```
1. public interface MyInterface {
2. }
3.
4. public class Base {
5. }
6.
7. public class Subclass extends Base implements MyInterface {
8. }
9.
10. public class OtherSubclass extends Base {
11. }
12.
13. public class NotASubclass implements MyInterface {
14. }
```

First off, it should be made clear that this is a valid code. An interface does not need to define any methods to be a valid interface. But the concepts we want to explain will work if there are methods, fields or if there are

none. Objects in OOP can have an “IS A” relationship with one another. For example, all the following statements are correct:

- **Subclass** IS A **Base**
- **Subclass** IS A **MyInterface**
- **OtherSubclass** IS A **Base**
- **NotASubclass** IS A **MyInterface**

This is the basis of polymorphism. A subclass can stand in place of a different object and replace the underlying implementation seamlessly. Let us show it with a more concrete example:

```
class Base {
public void hello() {
System.out.println("Hello");
}
}

class Subclass extends Base {
public void hello() {
System.out.println("Subclass");
}
}

Base subclass = new Subclass();
subclass.hello();
```

This code will print “Subclass”. Notice that since **Subclass** is A **Base**, we can treat it as such. The **hello** method is overridden; in this case, we did not use the **@Override** annotation to demonstrate a point. But it is indeed the best practice to add that.

Polymorphism is the ability to treat a class based on its type; we can use an interface for a type and “code to an interface”. This is a common best practice, as it lets us swap the implementation completely under the hood. A good example would be the built-in collection classes in Java’s util package.

```
void printAll(List<String> list) {
for(String element : list) {
System.out.println(element);
}
}
```

```
6.
7. void test() {
8. List<String> strings1 = new ArrayList();
9. strings1.add("1");
10. strings1.add("2");
11. strings1.add("3");
12. List<String> strings2 = List.of("A", "B", "C");
13. method(strings1);
14. method(strings2);
15. }
```

Notice that **printAll** is oblivious to the fact that it got two completely different types of lists. They both behave exactly the same. When invoking a method on the **List** interface, the Java compiler generates a virtual call which will be mapped at runtime to the right method implementation. This is seamless for us as developers and allows us to swap implementations under the hood.

Notice that in the code, we used a for loop to go over the elements of a **List**. This is a special version of the **for** loop that we can use to iterate collections and arrays easily. We could also use a stream to do that. We will discuss those shortly.

Built-in types

In addition to object and class, which we have already discussed to some extent, Java also has a few other important built-in types. Most notable would be primitive types such as **int**, which we discussed before. Each primitive type has a corresponding object type that can be used in cases where an object is needed. These corresponding object types are often known as primitive wrappers. In current implementations of Java, primitive types have a significant performance advantage over wrappers. With some synthetic benchmarks, the scale is at 20× the performance difference. This might change with the work done on project Valhalla, which we will discuss in *Chapter 3: 8 to 21 to GraalVM*.

Table 1.1 primitive types and their sizes. Note that all primitives in Java are “signed”, which means that they range between negative and positive numbers. The exception to this is **char**, which is not a number:

Primitive	Wrapper class	
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

create, an implicit **Class** object is created. But it also means another one is created to represent the array of our type. This is better illustrated with the following code:

```
1. // Good Usage
2. var first = new Fraction(1, 2);
3. var second = new Fraction(2, 3);
4. var result = first.add(second);
5. System.out.println(first + " + " + second+ " = " + result);
6.
7. var third = new Fraction(1, 2);
8.
9. // will throw an exception...
10. var forth = new Fraction(2, 0);
11. var secondResult = third.add(forth);
12. System.out.println(third + " + " + forth+ " = " + secondResult);
```

Notice a few things about this code. We can use the == operator to compare the pointers of the Class objects instead of using the **equals** method. There is one **Class** object per type, and we can rely on that. The object and array class objects are different object types, so in Lines 7 and 8, we create two arrays of different sizes. Notice they follow the same rules as regular objects as we see in Line 11.

This is where things get a little weird. Line 16 will generate a compilation error. The same problem will happen if we try to compare classes from an object and an array object. Both are of type **Class**, so why does the compiler not accept them? To understand the answer, we need to understand generics. The signature of the **getClass()** method is as follows:

```
1. public final Class<?> getClass()
```

The question mark symbol in the class plays a crucial role in clarifying the type of valid assignments and catching situations that do not make sense in the compilation stage. We will dig into this in the next section.

Generics and Erasure

When Java was initially developed, we did not have generics. We would write a list of strings using code similar to the following:

```
1. List stringList = new ArrayList();
2. stringList.add("My String");
3. String myString = (String)stringList.get(0);
```

This code would still compile and work just fine in Java 20, which is a testament to the amount of work the Java team puts into compatibility. Notice on Line 3 we need to cast the value returned by the get method. When we do not explicitly specify that the elements are of a specific type, we end up with the lowest common denominator: **Object**. Since all objects in Java derive from **Object**, we can just use that when passing something generic. This works really well, but as Java 5 came out, generics were added. The reason is simple; look at the following code:

```
1. List stringList = myApi.getList();
2. stringList.add("My String");
3. stringList.add(3);
```

This is problematic. If we have an API that returns a **List** or a similar object, we have no way to know what object types are expected other than the documentation. With generics, we can explicitly indicate the type of the list using **List<String>**, which will cause Line 3 to fail in the compiler. Since this logic was added after Java was completed, the generics are tacked onto Java. That means the JVM ignores them for the most part and does not really know about them, which makes the following code legal:

```
1. List<String> stringList = new ArrayList<>();
2. stringList.add("My String");
3.
4. List blank = (List)stringList;
5. blank.add(3);
6.
7. // Prints [My String, 3]
8. System.out.println(stringList);
9.
10. // Prints 3
11. System.out.println(blank.get(1));
12.
13. // Throws ClassCastException: Integer cannot be cast to String
14. System.out.println(stringList.get(1));
```

Since generics are erased in runtime, we can downcast the list object and remove the generic aspect. We can then add anything we want to the list. This works, and the list does indeed contain that information within it. The code will not compile if we try to add a number to the **stringList**. But this is where it gets “weird”.

In Line 11, we print the second element, and it is indeed the three that we added. However, if we print through the string list, we will get a runtime exception. We cannot cast an integer to a string. The reason for this is simple. In Line 8, we invoke the **toString()** method implicitly. This method does not get a type and is relatively generic. It just works as we would expect. The get method returns the type of the generic, but with an erasure like blank, it would return **Object**, so it would work. With the **stringList**, it would return a **String**; this fails since casts are checked in runtime. Java is a dynamic language and verifies such behaviors in the runtime.

As a side note, notice Line 1 in the code. You will notice the diamond operator < >, which we have near the end of the line. This is a syntax that was added in Java 8. See the following code to understand the various ways of writing this line:

```
1. // Original Java 5 Syntax
2. List<String> stringList = new ArrayList<String>();
3.
4. // Java 8 Diamond Operator
5. List<String> stringList = new ArrayList<>();
6.
7. // The var keyword can't use Diamond since the generic is needed
8. var stringList = new ArrayList<String>();
```

A simple generic can be declared easily within the class declaration code shown as follows:

```
1. public class MyObject<T> {
2. T value;
3. public T getValue() {
4. return value;
5. }
6. public void setValue(T value) {
7. this.value = value;
8. }
9. }
10.
11.
12. MyObject<Integer> obj = new MyObject<>();
13. obj.setValue(3);
```

This is pretty basic stuff, but there are some more elaborate cases. Let us

say we want the class to represent arithmetic expressions. All numeric wrapper objects derive from the **Number** class. Generics can represent this using wildcards such as **MyObject<? extends Number>**. We can also represent the flip side of such a relationship. Say we have an API that accepts an **Integer** but might work with one of its base classes (**Number** or **Object**); we can use the syntax **MyObject<? super Number>**.

Generics can be applied to methods as well. We will use the following example to show off the challenge of generics:

```
1. public class CompareNumbers {
2. public static <T extends Comparable> T max(T a, T b) {
3. return a.compareTo(b) > 0 ? a : b;
4. }
5.
6. public static void main(String[] argv) {
7. System.out.println(max(3, 5));
8. System.out.println(max(3.0, 5));
9. }
10. }
```

Comparable is a generic interface in Java that lets us compare two objects, returning whether one is smaller, equal, or larger than the other. This code defines T as a generic type, and we can then invoke max on both sets of arguments. Unfortunately, the output is as follows:

5

Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.Double (java.lang.Integer and java.lang.Double are in module java.base of loader 'bootstrap')

at java.base/java.lang.Double.compareTo(Double.java:159)

at com.debugagent.ch01.generics.CompareNumbers.max(CompareNum

at com.debugagent.ch01.generics.CompareNumbers.main(CompareNu

The first comparison worked just fine, but since the second comparison tried to compare an Integer and a Double, it failed. If we look at the compiler warnings for that code, we see that it already had some notion that something was not right here:

Raw use of parameterized class 'Comparable'

Unchecked call to 'compareTo(T)' as a member of raw type 'java.lang.Comparable'

These two issues refer to Lines 2 and 3 of the previous code, respectively. Comparable is a generic type, and we need to pass a generic to it. Fixing the code to add a <**T**> to the comparable interface solves the problem:

1. public static <T extends Comparable<T>> T max(T a, T b)

Once we do that, the second call to max will fail with the error:

'max(T, T)' in 'com.debugagent.ch01.generics.CompareNumbers' cannot be applied to '(double, int)'

This explains the problem clearly: why it fails at compile time instead of runtime.

Debugging

One of the best ways to learn a programming language is through the debugger. The REPL can be powerful, but it does not provide the conveniences of the IDE, and neither does it let you inspect variables. A debugger lets us see the inside of our application as we are working on it. This is tantamount to a mechanic inspecting a cylinder from inside the engine while it is running.

When we want to learn a new project or new platform, we can pick up the debugger and start verifying our assumptions. This is a powerful exercise that helps verify one's understanding of the subject matter. A program is a set of instructions and declarations. The debugger lets you see how they are carried out, which is often easier to understand. The same is true for a programming language. We can look at a method all we want, but as we step through it and inspect the changes, we can truly understand it.

Debugging Java

The debugger in IntelliJ/IDEA is one of the most powerful debuggers around. We will not go into its full set of capabilities, but let us go over its basic usage. We can set a breakpoint in IntelliJ/IDEA by clicking within the "gutter area" to the left of the editor. This places a red marker, as seen in *figure 1.8*. We can launch the debugger either via the right-click context menu (on the main class) or via the toolbar next to the run command.

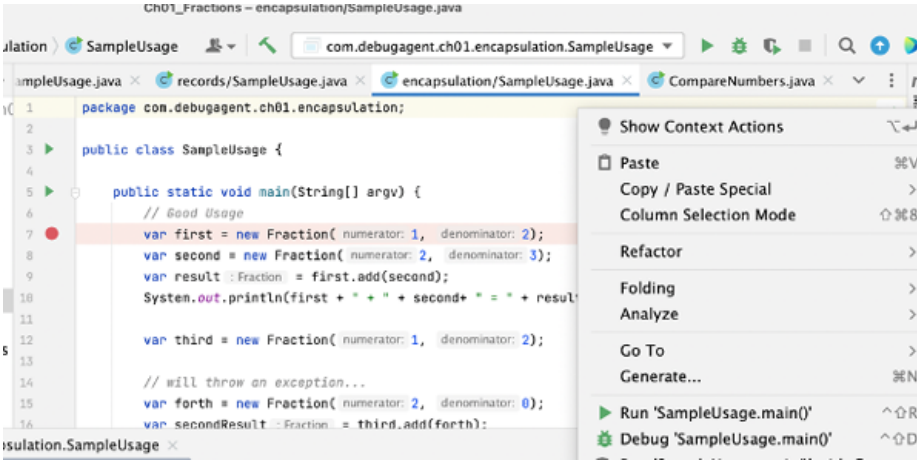

Figure 1.8: *Set a breakpoint and debug*

When the debugger reaches the breakpoint line, it is stopped on that line, and it presents us with information about the currently executing application. In the editor window, we can see the line we are currently debugging, as shown in *figure 1.8*. We can step over to the next line or step into a method call. This way, we can traverse one line at a time until we understand the root cause of the issue.

On the bottom left, we can see the stack trace for the current position in the code. It includes the method call chain. It is clickable and will take us back through the stack frames. This way, we can inspect the various stages that led us to this point. On the right side, we can see the current state and the values of the variables along the way. But it is a good place to start practicing in the debugger instead of theory alone.

Conclusion

In this chapter, we refreshed our basic understanding of modern Java syntax. We also learned some newer concepts in Java, such as records, var keyword, and so on, but we are still in the introductory phase of understanding the basics. We talked about the JVMs separation between types, generics, erasure, and much more.

We then briefly discussed the concepts of IDE-based debugging. While you do not need an IDE to work with Java, its power truly shines when you take advantage of an IDE. As such, we discussed IntelliJ/IDEA and also IDE-based debugging, which is a crucial skill that is not present enough in the common curriculum.

We discussed OOP in this chapter, but it was a brief overview. In the upcoming chapter, we will delve into the core concepts and try to understand what makes it a powerful paradigm for solving problems.

Points to remember

The three principles of OOP are as follows:

1. Encapsulation
2. Inheritance
3. Polymorphism

Of those three, encapsulation is the simplest and arguably the most important capability. Java has two basic type groups:

1. Primitives
2. Objects

There are eight primitive types, and each one of them has a primitive wrapper:

1. boolean
2. byte
3. char
4. short
5. int
6. long
7. float
8. double

Everything else is an object. Including arrays. Objects can be customized with generic types, but those types are removed in runtime due to erasure that can lead to unexpected behaviors, such as the ability to add a string to a list of integers.

Multiple choice questions

A. Which of the following are primitive types:

1. int
2. String
3. Class
4. byte[]
5. char
6. <T>

B. What is the difference between class and class?

1. They are synonyms for the same thing
2. You can have many Class objects but only one class
3. class is a keyword Class is an object
4. Class is a wrapper for the class primitive

C. Which of the following is a true statement

1. int[] IS A Object
2. Object IS A Class
3. int IS A Integer
4. BaseClass IS A Subclass

D. Erasure means that:

1. There is no checking on generic objects ever
2. Types are completely unknown at runtime
3. Generics need to be very specific object types so the compiler will know how to keep them
4. Generic types are removed during compilation and replaced with casts

Answers

A. 1, 5

B. 3

C. 1

D. 4

---

1 https://foojay.io/almanac/jdk-20/

2 https://sdkman.io/

3 https://www.jetbrains.com/idea/download/

4 The complete projects are available here: https://github.com/shai-almog/java-book

5 https://en.wikipedia.org/wiki/List_of_Java_keywords

6 This is for non-native English speakers; I did not learn the English terms in school and had to look them up

7 https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
OOP Patterns

Introduction

Java is a classic object-oriented language. As such, it contains many common OOP patterns. The term patterns gained traction in the early OOP community and was used to describe common best practices and approaches. This culminated in the **Gang of Four** (**GoF**) Design Patterns book, which is a bit out of date by now, but there is a more current website[1].

In this chapter, we will explore the best ways to leverage OOP, what we would consider as good design, and what type of OOP principles are expressed in the Java API.

Structure

In this chapter, we will discuss the following topics:

- Why OOP?
- Basic Object-Oriented Design
- Common patterns
    - Singleton
    - Factory
    - Builder
    - Adapter
    - Façade
    - Proxy
    - Observer
    - Command
    - Iterator

- Immutability
- Functional programming

Objectives

By the end of this chapter, the reader will understand the value of working with OOP. Where OOP makes sense and why Java is not a pure OOP language. We will discuss the value of functional programming and how Java incorporated some of those ideas into the language and the API.

Why OOP?

We discussed the power of encapsulation, inheritance, and polymorphism in the first chapter. But OOP goes well beyond those core principles. It is a design philosophy and a structuring system that binds the data with the operations that process the data. Before OOP, we had memory structures, and arbitrary functions could modify any arbitrary object. These functions were independent of one another and did not have a relationship between them. Thanks to OOP, we can include the data with the application operations in a single encapsulated entity. This is easier for us when we build the system; we can enclose everything in black boxes and understand complex relationships without knowing the underlying details.

It is also easier for the user of an OOP API. Operations are encapsulated and have all the data they need. It is a powerful concept that we can leverage to build sophisticated applications. The main challenge, however, becomes the modeling.

In the 90s, the author worked on a flight simulator built in C++. The code was a derivative of an older simulator whose roots were in the early days of C++. Due to its age, the developers did not know how to design an object-oriented API. Every aircraft used multiple inheritances to derive all the capabilities of the plane, for example, motion, display, and so on. This design created many problems, and it was replaced with composition. Composition means holding a pointer to a different object rather than inheriting its capabilities. This is one of the first rules of object-oriented design: “composition is usually better than inheritance”.

It is also considered a best practice to code to an interface; a good example would be to use **List** instead of **ArrayList**. The problem is that these best practices can be taken too far. A colleague of the author worked on a project where the lead engineer took this concept to the extreme. He prohibited inheritance entirely and forced everyone to use interfaces and composition. The project was a mess of boilerplate code that was copied and pasted all over the place to facilitate common patterns. Some developers try to avoid inheritance because it is rigid and inflexible. This is

true, but so are the foundations on which we build our houses. We need rigid foundations to build a solid design. Inheritance is good when done right. We need to be sure of it and need to make a conscious choice.

Designing the right hierarchy is challenging. We make mistakes, and they are sometimes hard to spot when we are drawing a design on a whiteboard. That is why design needs to be flexible and adaptable—so that we can refactor the hierarchy if needed. Java IDEs have some of the best refactoring tools in the industry. They are Java's secret weapon.

Basic Object-Oriented Design

In **Object-Oriented Design** (**OOD**), we outline the architecture of an object-oriented system. In some cases, this is done ad hoc, and in others, it can be an elaborate process rich with **Unified Modeling Language** (**UML**) diagrams. There are many books written on the subject and many approaches. We will focus on the high-level ideas behind a design, what purposes it serves, and how we can interpret it when writing code.

In *figure 2.1*, we can see a UML diagram representing a portion of the design of the Spring Boot pet clinic. This type of diagram represents a class hierarchy for a pet clinic application. The diagram shows all these classes derive (inherit) from the **BaseEntity** class. That class includes common functionalities in all elements, specifically an ID. Notice that this diagram is partial since it does not include properties and does not show composition. For example, the owner includes pets. This is typically represented by a composition line in UML.

Figure 2.1: *Partial UML diagram of the Spring Boot Pet Clinic Demo*

We also have a person hierarchy, which is used to represent three distinct objects. As we build a system, we will often find that our hierarchy

includes common functionality and requires similar data types. At this point, it will make sense to pull the functionality up in the hierarchy and add a base class to consolidate common features.

It is very difficult to get a design “right” the first time around. We need to iterate and adapt on design to find the right set of tradeoffs. A common mistake is to “over design” and try to create a solution that is overly generic. These design faults are hard to quantify, and you will develop design sensibilities as you are exposed to well-designed systems and gain experience.

Common patterns

The Java API and applications built on top of it feature many common design patterns and conventions. Some of them represent original GoF patterns, whereas others are uniquely Java patterns. As the popularity of patterns grew, corresponding anti-patterns trends emerged. These are patterns that are typically considered bad practices.

The true value of patterns is linguistic. We can describe complex behaviors by simply bringing up the pattern. These behaviors are often language agnostic and help us build a mental model of a system without a detailed understanding of the specific mechanics involved.

Singleton

The Singleton is probably the most universally recognizable pattern. It is also commonly considered an anti-pattern. Despite that, it is widely used; we can see it in the Java API in classes such as **Desktop** or **Runtime**. Let us start by defining the simplest Singleton possible:

```
1. public class SimpleSingleton {
2. private static final SimpleSingleton INSTANCE = new SimpleSingleton();
3.
4. private SimpleSingleton() {}
5.
6. public static SimpleSingleton getInstance() {
7. return INSTANCE;
8. }
9.
10. public void doSomething() {
11. // code
12. }
```

```
13. }
```

A Singleton is a class for which we can have only one instance. This is roughly the equivalent of having a class where all the methods are static. Notice that the instance of the class is created statically in this case, and the constructor is private. That means the code that will try to create an instance of **SimpleSingleton** will fail. The only way to get the instance is through the **getInstance()** method.

Why do we need this pattern if we can just use static methods?

The main benefit of a singleton is in its ability to evolve as an API. Following is a more realistic example of a singleton. In this case, the instance of the class is specific to the platform we are running on. In this way, our class can provide different functionality when running on Windows or on a Mac. This would be seamless to the rest of the code:

```
1. public abstract class PlatformSingleton {
2. private static final PlatformSingleton INSTANCE;
3.
4. static {
5. if(System.getProperty("os.name").toLowerCase().contains("mac")) {
6. INSTANCE = new Mac();
7. } else {
8. INSTANCE = new Windows();
9. }
10. }
11.
12. public PlatformSingleton getInstance() {
13. return INSTANCE;
14. }
15.
16. public abstract void platformMethod();
17.
18. static class Mac extends PlatformSingleton {
19. @Override
20. public void platformMethod() {
21. // mac implementation
22. }
23. }
```

```
24.
25. static class Windows extends PlatformSingleton {
26. @Override
27. public void platformMethod() {
28. // windows implementation
29. }
30. }
31. }
```

There are several interesting elements in the code. Notice we do not initialize the final variable in Line 2. This is illegal in Java since the final is a constant field and must be initialized. There is one special case for that, and it starts in Line 4. A static final field can be initialized in the static block. A static block executes when the class is loaded and can implement logic such as field initialization and so on. Notice that the **INSTANCE** variable can be assigned only once. The compiler will check that it is initialized only once and never changed in every path within the static initializer.

Also, notice that the class is **abstract**; this lets us define a class that does not implement all its methods. You cannot create an instance of an abstract class. That is why we no longer need the private constructor trick. At the bottom of the class, we have two inner classes that extend it. Inner classes are a convenient feature of Java that lets us define a class within another class or within a method. In this case, the inner classes are declared static, which means they do not have a parent dependency.

By default, an inner class can reference the instance of its parent class. This means we can do something like the following:

```
1. public class Outer {
2. private int field;
3.
4. class Inner {
5. public void innerMethod() {
6. field = 3;
7. }
8. }
9. }
```

The **Inner** class has a reference to the instance of the surrounding **Outer** class. That is how it can resolve the lookup to the field. The reference to the **Outer** class can be referenced explicitly using the name of the outer

class and this example: **Outer.this**. By defining the inner class as static, we disable that capability since the inner class does not need a reference to the enclosing instance.

But there is still a major limitation. We create the instance statically as the class loads, which can be very early in the application, and we might not need the class at all. We might need sophisticated logic to initialize the class. In those cases, we might want to create instances lazily, such as the following:

```
1. public class LazySingleton {
2. private static LazySingleton INSTANCE;
3. private LazySingleton() {}
4.
5. public LazySingleton getInstance() {
6. if(INSTANCE == null) {
7. INSTANCE = new LazySingleton();
8. }
9. return INSTANCE;
10. }
11. }
```

This can work fine. It will have a slight overhead of null checking every time we invoke it, but that is fine. Unfortunately, it might have another problem. Java lets us create multiple threads of execution. This is something we will discuss in-depth in *Chapter 3: 8 to 21 to GraalVM* and *Chapter 4: Modern Threading*. But for now, let us just explain that a thread is a different execution stack running concurrently with the current stack. Java's built-in keyword **synchronized** allows only one thread into the **synchronized** block. We can apply it to this object or to an arbitrary object. It is a common pattern to dedicate a specific LOCK object on which to synchronize to avoid potential side effects.

Unfortunately, **synchronized** is slow. So, we do not want to surround the whole **getInstance()** method body with a **synchronized** context. There is a trick that is very common for singletons, and it is called double-checked locking, as demonstrated in the following code block:

```
1. public class LazySingleton {
2. private static LazySingleton INSTANCE;
3. private static final Object LOCK = new Object();
4. private LazySingleton() {}
5.
```

```
6. public LazySingleton getInstance() {
7. if(INSTANCE == null) {
8. synchronized (LOCK) {
9. if(INSTANCE == null) {
10. INSTANCE = new LazySingleton();
11. }
12. }
13. }
14. return INSTANCE;
15. }
16. }
```

In Line 7, we check if the instance is not null. Assuming it is not null, we can just return. If it is null, then we might need to initialize it. That is the case where we need to synchronize the access since other operations that just return the variable are thread-safe. By placing the **synchronized** block within the **if** block, we remove its overhead for the most common use case. This is where things get tricky. We check again if the instance is null. Why?

Imagine two threads entering the method together, and the first thread checks if the instance is null, which it is. Then that same thread starts checking if it can enter the synchronized block. The second arrives just then, and as the instance is still null, it too arrives at the synchronized block. One thread enters the synchronized block. Since the instance is still null, it gets initialized and returned.

The second thread is released and can now enter the synchronized block too. But by now, the instance has a value. While the first **if** statement was **true**, the second **if** statement ended up as **false**.

This is all cool and powerful, so why do people consider singletons an anti-pattern?

There are several problems, but the most important one is the problem with wiring. Singletons are very hard to replace when we do a test. If we use a class or an interface instance, we can usually replace the instance with a fake instance so we can test it more easily. A Singleton is usually deeply bound to the code and cannot be mocked easily. We will discuss mocking more in-depth when we talk about testing in *Chapter 6: Testing and CI*. *Figure 2.2* is a UML diagram representing the singletons we discussed in this section.

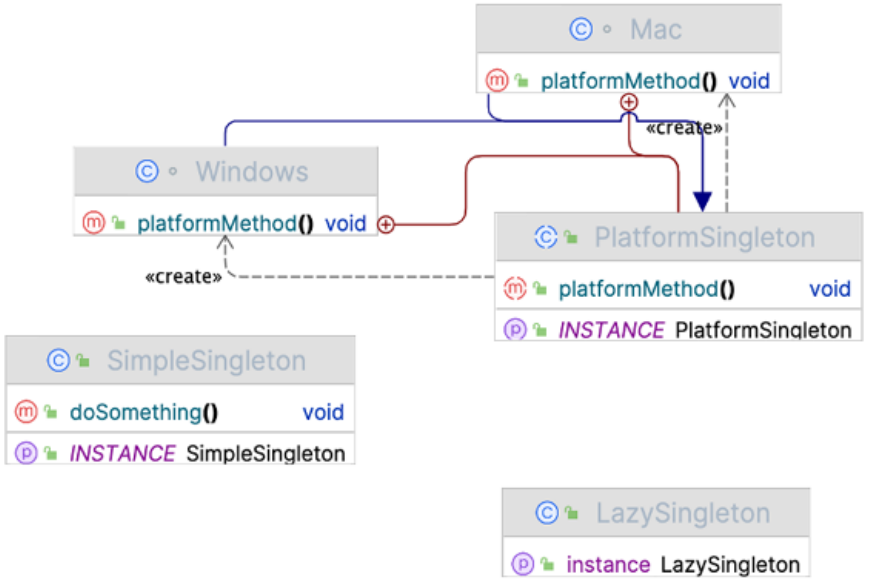

Figure 2.2: *UML diagram of the Singleton patterns discussed in this chapter*

Factory

The factory pattern has some variations, such as method factory and abstract factory. Most developers just use the term factory to describe all of them, as the difference is relatively nuanced. Why do we need a factory?

Why not invoke a **new** to create an object instance? In some cases, the new call will not work well. A good example would be a UI widget. Let us say we want to add a button to the screen. If we are running on iOS, we would want it to look in a particular way. But if we are running on Windows, we would expect something completely different. By creating a button using a factory, we can provide different implementations of the buttons to suit the various needs. This lets us abstract the underlying implementation and make that work seamlessly with the rest of the code. Let us discuss the following three common factory types:

1. **Factory**: A class that has a method or multiple methods that create objects. In Java, the collections class acts as a factory.
2. **Factory Method**: It creates an object in a way that can change due to polymorphism; it typically returns a base class instance that can be any concrete implementation. **Calendar.getInstance()** is such a factory method.
3. **Abstract Factory**: It is an interface that we use to create instances of the objects we need. This appears in the Java API in.

This is a bit abstract. Let us discuss these various options with some concrete examples. Here we have a simple factory:

```
1. public class Factory {
2. public CreatedClass createInstance() {
3. return new CreatedClass();
4. }
5. }
```

A class is created via a method. Typically, it would be an instance method, but it can be static, as is the case in the **Collections** class in Java. This lets us separate the logic from the constructor. The constructor is sometimes problematic to use as it cannot return a different result. Furthermore, throwing exceptions from constructors is considered a bad practice. It creates a situation where all subclasses might throw an exception. By using any type of factory, we remove that code from the constructor and eliminate the inheritance problem.

The factory method is just as simple, with one small nuance:

```
1. public class FactoryMethod {
2. public AbstractBase createInstance() {
3. return new CreatedClass();
4. }
5. }
```

The difference is subtle. We have a different type in the return value and can adapt the behavior based on the class instance. We can have multiple implementations subclassing the **FactoryMethod** class, and each can return a different subclass of **AbstractBase**. That is a very powerful tool.

The abstract factory goes further. It is an abstract class or an interface whose implementation is private or unknown to us. We can often obtain an instance of the abstract factory using tools such as dependency injection or dynamic class loading. This makes it harder to use, but it allows the maximum amount of flexibility:

```
1. public interface AbstractFactory {
2. AbstractBase createInstance();
3. }
```

In *figure 2.3*, we show the factory implementations discussed in this section.

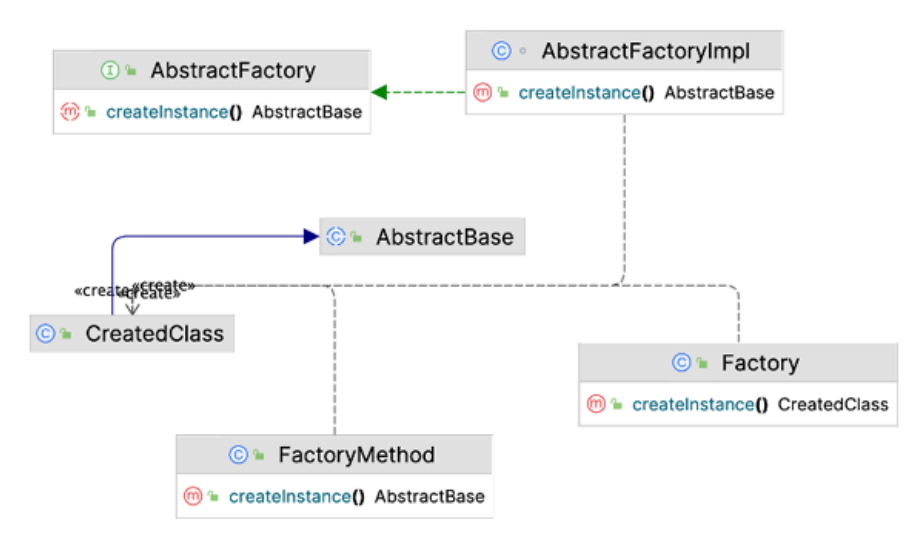

Figure 2.3: *UML diagram of the factory patterns discussed in this chapter*

Builder

Builder is another pattern for constructing an object. The builder is designed for fluent creation syntax instead of a compiler.

1. Instructor ShaiAlmog = Instructor.builder()
2. .withProfessionalExperience(30, TimeUnit.YEARS)
3. .withTopCompanies("Sun", "Oracle", "Codename One", "Lightrun")
4. .withMastodon("@debugagent@mastodon.social")
5. .withGitHub("github.com/shai-almog")
6. .withLinkedIn("linkedin.com/in/shai-almog-81a42/")
7. .withBlog("debugagent.com")
8. .withBlog("talktotheduck.dev")
9. .withTwitter("twitter.com/debugagent")
10. .build();

This is a builder pattern. I could write similar code with a constructor by invoking **new Instructor(...)**, but in this case, the code is so clear I could include it on a slide, and it would be clear to a person in the audience. A builder can take many forms, and one form is by returning a modified instance of the same class (as **StringBuilder** does in Java) or by building a class using a separate dedicated class. For example, the simpler version of the builder patterns can be used like the following:

1. StringBuilder str = new StringBuilder("The value of 'i' is ")
2. .append(i)

```
3. .append(" and the value of x is ")
4. .append(x);
```

A more elaborate implementation of the builder pattern can be seen in the following code. Notice that it is very verbose, but it lets us customize every stage of creation with very fluid code:

```
1. public class Created {
2. private int val1;
3. private String val2;
4. private boolean val3;
5.
6. private Created() {
7. }
8.
9. public static Builder create() {
10. return new Builder();
11. }
12.
13. public static class Builder {
14. Created c = new Created();
15.
16. private Builder() {}
17.
18. public Builder val1(int val1) {
19. c.val1 = val1;
20. return this;
21. }
22.
23. public Builder val2(String val2) {
24. c.val2 = val2;
25. return this;
26. }
27.
28. public Builder val3(boolean val3) {
29. c.val3 = val3;
30. return this;
```

```
31. }
32.
33. public Created build() {
34. if(c.val2 == null) {
35. throw new IllegalStateException();
36. }
37. return c;
38. }
39. }
40.
41. public static void main(String s) {
42. Created created = Created.create()
43. .val1(1)
44. .val2("Test")
45. .val3(false).build();
46. }
47. }
```

In Line 42 and onwards, we can see the usage of the API, which has advantages over a constructor with nameless arguments. Notice we can keep the state of the class completely hidden outside of the builder and thus force deeper state encapsulation thanks to this approach. This is made obvious in *figure 2.4,* where we show the UML diagram representing this code.

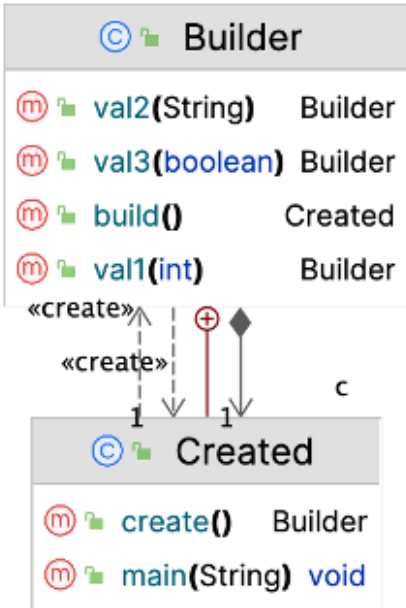

Figure 2.4: *UML diagram of the builder pattern discussed in this chapter*

Adapter

An adapter takes one object type and makes it seem like a different object type. A great example in Java would be the **Arrays.asList()** method which accepts an array object and returns an implementation of the **List** interface. It adapts one object type for another, so we can pass an array to an API that accepts a **List** by wrapping it in an adapter. The following code implements an adapter that adapts between a **Runnable** and an **ActionListener**. This lets us send an event to a runnable:

1. public class CallableAdapter {
2. public ActionListener callableAdapter(Runnable runnable) {
3. class Act implements ActionListener {
4. @Override
5. public void actionPerformed(ActionEvent e) {
6. runnable.run();

```
7. }
8. }
9. return new Act();
10. }
11. }
```

Notice that we declared the inner class directly within the method. Also, notice that because it is declared in the method itself, it has access to fields in the scope of the method, such as the runnable field. An inner class can only access fields that are final or effectively final. A field that is changed in any place cannot be accessed in the inner class.

This syntax is very verbose, and the inner class takes up a lot of boilerplate. Thankfully, Java provides a shorthand syntax. The anonymous inner class. The name of the class is not necessary, and neither is the **class** keyword. By adding curly brackets after the creation of the object, we create an inner class implicitly without a regular name. The JVM names such classes with the parent class name followed by a $ sign and a number. The code for an anonymous inner class looks like the following:

```
1. public class CallableAdapter {
2. public ActionListener callableAdapter(Runnable runnable) {
3. return new ActionListener() {
4. @Override
5. public void actionPerformed(ActionEvent e) {
6. runnable.run();
7. }
8. };
9. }
10. }
11. }
```

This is much better. We removed the class declaration and the implements clause. We no longer need to come up with a name, but this still includes a lot of boilerplate. The compiler knows that we will return an **ActionListener**. Why do we need to write **new ActionListener()**? It knows the interface only has one method; why do we need the **actionPerformed** declaration? That is where Java 8 introduced Lamdas. They go a bit further than just “shorthand for anonymous inner classes”, but that is a major aspect related to them. This is how the same code looks using a Lambda:

```
1. public class CallableAdapter {
```

```
2. public ActionListener callableAdapter(Runnable runnable) {
3. return e -> runnable.run();
4. }
5. }
```

Since we only have one parameter, we can even drop the type declaration and leave the name only. The compiler does everything else.

Façade

The Façade is not very common in the core Java API. In this pattern, a class or interface serves as an intermediate through which we hide an underlying implementation that is more complex. It is a very simple pattern. The main benefit of using a façade is future flexibility. By inserting a façade before the actual API, we are adding a layer of indirection that reduces coupling between the layers and lets us customize API behaviors. Following is a simple façade implementation:

```
1. public class Facade {
2. private final InternalAPI1 internalAPI1 = new InternalAPI1();
3. private final InternalAPI2 internalAPI2 = new InternalAPI2();
4. private final InternalAPI3 internalAPI3 = new InternalAPI3();
5.
6. public void api1() {
7. internalAPI1.api1();
8. }
9.
10. public void api2() {
11. internalAPI2.api2();
12. }
13.
14. public void api3() {
15. internalAPI3.api3();
16. }
17. }
```

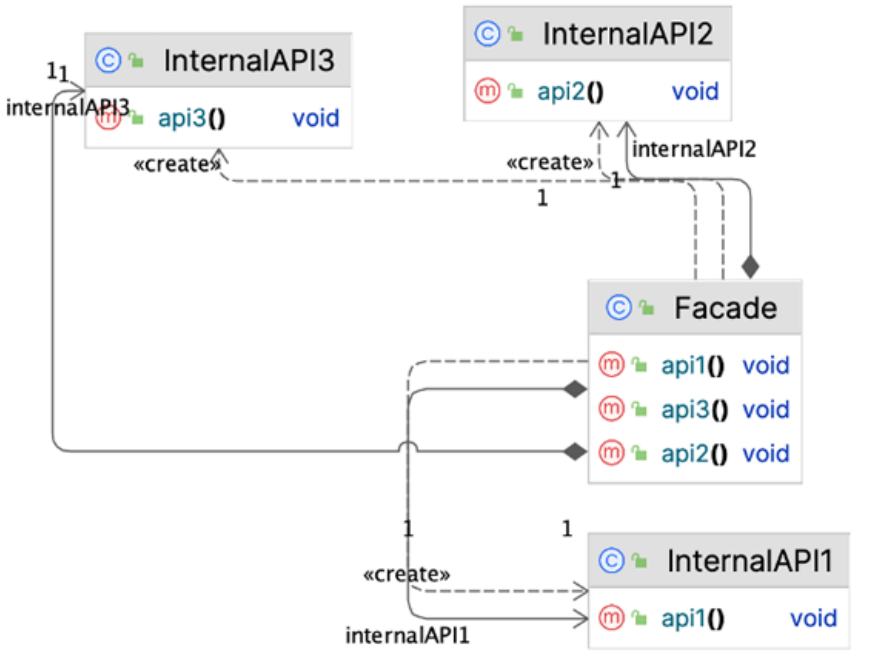

Figure 2.5: *UML diagram of the Façade*

Proxy

Façades are often mistaken for proxies, and in some sense, they are. A proxy can sometimes be a façade inspected from a different angle. A proxy is a class that implements the interface of another class, and it pretends to be that other class. It then delegates the work to that other class while doing "interesting things" in between.

A good example is a caching proxy that detects if a method is invoked with the same arguments and returns a cached value. Java has a built-in Proxy API that lets us implement interfaces dynamically using reflection. We will discuss that soon enough. But first, let us discusses the simplest type of proxy we can build. The following code implements a proxy that checks whether the underlying runnable class can be invoked, and it only invokes it if the permission flag is set to true. This sort of proxy lets us enforce global behavior without coding it into every runnable class:

1. public class SimpleSecurityProxy implements Runnable {
2. private Runnable internal;
3. private boolean permission;
4. public SimpleSecurityProxy(Runnable internal) {
5. this.internal = internal;

```
}

public void setPermission(boolean permission) {
this.permission = permission;
}

@Override
public void run() {
if(permission) {
internal.run();
}
}
}
```

This is great, but what if we wanted to do something like that on a more general level? What if we wanted to apply security constraints to every interface? We would need to write proxies that implement all the interfaces around. Luckily, Java has the reflection functionality. This set of APIs lets us introspect a class at runtime and invoke methods, read fields, and even implement interfaces dynamically. This is a very powerful tool as it lets us build very generic code. It is also somewhat problematic as code written in that way is awkward, hard to debug, and slow. Still, it is a very powerful tool. The previous code can be implemented using reflection for any interface type like the following:

```
public class GenericSecurityProxy {
private boolean permission;
public void setPermission(boolean permission) {
this.permission = permission;
}

public Object create(Object wrappedObject, Class<?> wrappedInterface) {
return Proxy.newProxyInstance(getClass().getClassLoader(),
new Class[]{wrappedInterface},
(proxy, method, args) -> {
if(permission) {
return method.invoke(wrappedObject, args);
}
```

```
14. return null;
15. });
16. }
17.
18. public static void main(String[] argv) {
19. Runnable r = () -> System.out.println("Print invoked");
20. GenericSecurityProxy g = new GenericSecurityProxy();
21. Runnable wrapper = (Runnable)g.create(r, Runnable.class);
22. wrapper.run();
23. g.setPermission(true);
24. wrapper.run();
25. }
26. }
```

There is a lot going on here, so let us review. In Line 7, we declare a method that wraps an arbitrary object. Notice we accept two parameters, the object we wish to wrap and the interface we need to represent. This might seem redundant, but why pass the class instance when we can simply invoke **wrappedObject.getClass()**?

The object would never be an instance of an interface. It will be an instance of an object, and the **Class** object corresponding to it will not be matched properly. In fact, if we would do that, the error would be as follows:

java.lang.IllegalArgumentException: com.debugagent.ch02.patterns.proxy.GenericSecurityProxy$$Lambda$14/0x0000000801003200 is not an interface

We need a class object representing the specific interface. If you look at Line 21, you can see that an instance of Runnable is passed to the method, and the **Runnable.class** syntax is used to get the **Class** object representing the class.

Back in Line 8, we see the usage of the Proxy class. The **newProxyInstance** method accepts three parameters:

- **ClassLoader**: This is an abstract class that we can override to control the way the JVM loads classes.
- **Array of Interfaces**: Since a **Proxy** can implement more than one interface, the method accepts an array as an argument. Notice the shorthand syntax to create an array with a fixed set of values.
- **InvocationHandler instance**: Since **InvocationHandler** is an

interface with a single method, we can convert the code to lambda. The callback returns the return value from the invocation and accepts three parameters:

- **Proxy**: The object instance.
- **Method**: An object representing the method itself in the reflection API. This is equivalent to the method signature and includes the name, parent class, return value, and arguments.
- **Arguments**: The values of the arguments passed to the method.

In the implementation of that Lambda, we check permission and then simply invoke the method on the underlying object instance, if applicable. At the bottom of the listing, you can see a sample usage of the API that prints the invocation line only once. Reflection is one of the most powerful APIs in Java, and the Proxy API is the basis for some amazing capabilities used by many toolchains. The proxy pattern and sometimes this very API is at the basis of frameworks like Spring, Microprofile, and so on.

Observer

The observer pattern sends a notification about a change or an event to a different object. The Java API has an Observer object, but the best-known example of an observer in Java is the ActionListener. To implement the observer, we need a listener interface and an implementation class. Let us start with the listener interface:

```
1. public interface EventListener {
2. void onEvent(Map<String, Object> eventMetadata);
3. }
```

The interface is not useful on its own. To make use of it, we need an observable component like the following. It is common to use an observable component like this as a base class or with composition:

```
1. public class Observable {
2. private List<EventListener> listeners = new ArrayList<>();
3. public void addListener(EventListener listener) {
4. listeners.add(listener);
5. }
6.
7. public void removeListener(EventListener listener) {
8. listeners.remove(listener);
9. }
```

```
10.
11. public void fireEvent(Map<String, Object> metadata) {
12. for(EventListener e : listeners) {
13. e.onEvent(metadata);
14. }
15. }
16. }
```

We can add an event listener to the observable object, and we can send an event to all the listening objects. This decouples dependency. A generic component does not need to know about the components that are interested in it. For example, we can build an observable button component, and it does not need to know about every click listener. It can just invoke **fireEvent** when it is clicked, and the appropriate listener will be invoked.

Figure 2.6 shows the observer pattern as a UML diagram.

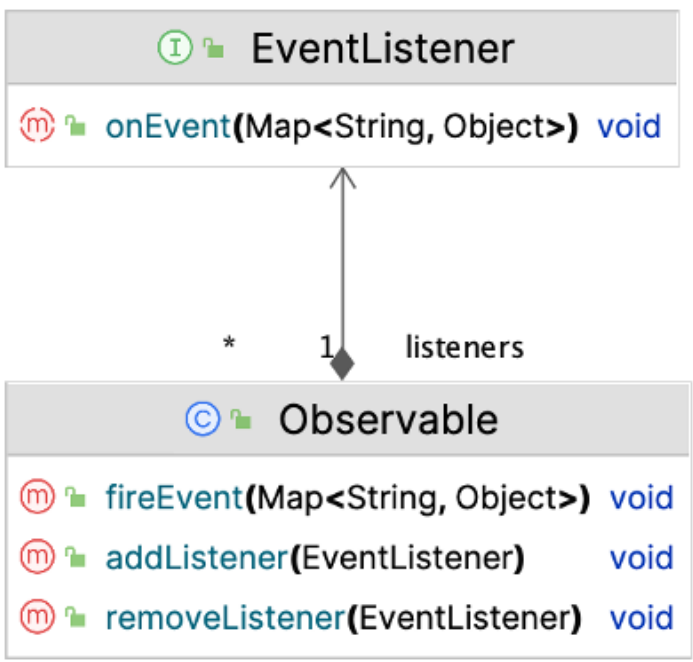

Figure 2.6: *UML diagram of the observer pattern*

Command

Undo is so common we practically take it for granted. In many cases, it is

implemented through the command pattern. In the command pattern, every operation is encapsulated in a command that includes all the elements required for the execution of the said operation. Every command can be performed and reverted. By adding commands to a queue and keeping performed commands in an undo queue, we can enable undo and redo. This is a non-trivial pattern, but the core concept is relatively simple. We start with the command, which can be an interface or an abstract base class:

```
1. public abstract class Command {
2. public abstract void perform();
3. public abstract void undo();
4. }
```

We would implement command subclasses with the full state required for performing the operation or undoing it. For example, if a command copies an item to a clipboard, we would include a reference to that item and to the previous content of the clipboard so that we can restore that. The command is added to a queue for processing and undoing:

```
1. public class CommandQueue {
2. private List<Command> performedCommands = new ArrayList<>();
3. private List<Command> undoneCommands = new ArrayList<>();
4. public void perform(Command cmd) {
5. cmd.perform();
6. performedCommands.add(cmd);
7. undoneCommands.clear();
8. }
9.
10. public boolean canUndo() {
11. return !performedCommands.isEmpty();
12. }
13.
14. public boolean canRedo() {
15. return !undoneCommands.isEmpty();
16. }
17.
18. public void undo() {
19. Command cmd =
```

```
    performedCommands.remove(performedCommands.size() - 1);
20. cmd.undo();
21. undoneCommands.add(cmd);
22. }
23.
24. public void redo() {
25. Command cmd =
    undoneCommands.remove(undoneCommands.size() - 1);
26. cmd.perform();
27. performedCommands.add(cmd);
28. }
29. }
```

We can perform a command via the perform operation in the queue. This adds it to the list of performed commands but, more importantly, clears the redo list. The redo process expects the application to be in a specific state. If we change the state and then try to redo a command that was undone previously, it might not work as expected.

Figure 2.7 shows the command pattern as a UML diagram.

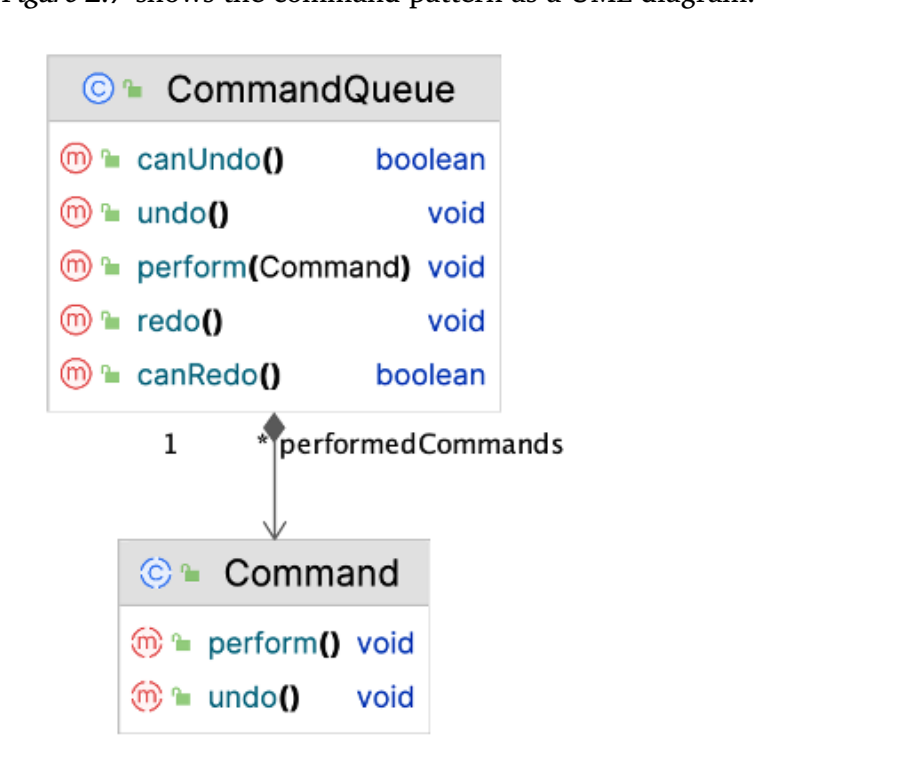

Figure 2.7: UML diagram of the command pattern

Iterator

The iterator pattern lets us traverse a collection of elements in a generic way that ignores the underlying implementation details. Java has an Iterator interface that implements that pattern rather nicely. You might have noticed in the past that we used syntax like the following to traverse a list of elements:

```
1. for(MyComponent cmp : list) {
2. cmp.doSomething();
3. }
```

Under the hood, the compiler has two supported types for the second argument. One is for array types. The other type is the **Iterable** interface, which is a part of the Java API. This interface provides an **iterator()** method that returns the Iterator instance. The iterator can move over the elements and get the respective value from each element. We can use it to traverse a list using code like the following:

```
1. List<String> list = List.of("Hi", "World");
2. var iterator = list.iterator();
3. while (iterator.hasNext()) {
4. System.out.println(iterator.next());
5. }
```

This is a very powerful tool but obviously not as useful due to the capabilities of the **for** loop and streams (which we will discuss soon). There are some edge cases where we are interested in deeper control where the iterator can be used directly. But there is a deeper reason we need to understand this pattern. The following code is a failscan; can you guess why?

```
1. List<String> list = new ArrayList<>(List.of("One", "Two", "Three"));
2. var iterator = list.iterator();
3. while (iterator.hasNext()) {
4. String current = iterator.next();
5. if(current.equalsIgnoreCase("one")) {
6. list.remove(current);
7. }
8. }
```

Notice that in the first line, we use an **ArrayList** instead of the results of **List.of()**. The reason for that is that **List.of()** returns an unmodifiable list.

In Line 6, we wish to remove an element from the list, and it would fail for an unmodifiable list. However, because we removed an element from the list, the iterator becomes invalid and will fail with a **ConcurrentModificationException**. Java developers run into that exception frequently and incorrectly assume that this relates to threading and concurrency. Usually, it is just code that modifies a collection as we traverse it. This impacts the for-loop syntax as well.

Java's iterator is aware of this problem and has a feature that is not part of the classic pattern. It has an **iterator.remove()** method, which we can use instead of Line 6 in the previous listing. That would solve the concurrent modification exception and let us remove elements dynamically. Adding elements or performing other modifications to the underlying store still is not possible; the solution is to create a new list with the modified content.

Immutability

Immutability is one of the most powerful concepts in programming. In an object-oriented language, an immutable object is an object that cannot be modified once it is created. In Java String, as well as all the primitive wrapper, objects are immutable. This lets us make many assumptions about these objects. This is sometimes confusing to developers new to Java, who would use APIs such as **str.toLowerCase()** and expect **str** to have a lowercase string. Instead, the method returns a new instance of the string with lowercase.

It is widely considered that immutability promotes better code, albeit the cost of performance. Immutability makes us think about every change we make. It makes it easy to track changes as atomic operations and reduces issues that can occur due to concurrency or invalid (partially modified) states. For example, say we have an object representing a date, as seen in the following code:

```
1. var mutableDate = new Date();
2. mutableDate.setMonth(2);
3. mutableDate.setDay(29);
4. mutableDate.setYear(2001);
5.
6. var immutableDate = Date.create(2001, 2, 29);
```

February 29th did not exist in 2001 because it was not a leap year. However, this would be very hard to test for the mutable date. Furthermore, one could just avoid invoking **setYear**, thus, allowing an invalid date altogether. The immutable date can test the validity of the call to create. We can also build it using the builder pattern to facilitate a fluid

syntax.

Unfortunately, Java's support for immutability is partially foiled by the language. At this time, Java does not have stack objects, and all non-primitive elements in Java are passed by reference. This will partially change due to Valhalla, as we discuss in the upcoming chapter, but it will not change in a way that will significantly impact the next bit of information. Let us say we have an implementation of **String** which is immutable. **String** has a **toCharArray()** method, which returns the characters within it. One might assume the code for that method in **String** looks roughly like the following:

```
1. final char[] chars;
2. String(final char[] chars) {
3. this.chars = chars;
4. }
5.
6. public char[] toCharArray() {
7. return chars;
8. }
```

But this is not the case. The **final** keyword in Java has the following three purposes:

1. **On a Class**: It means the class cannot be inherited. **String** itself is final. There are security reasons behind that. It prevents someone from modifying or changing the behavior of a String and gaining undue access to the system.
2. **On a Method**: It prevents a user from overriding said method.
3. **On a Field or Variable**: It prevents modification to the field. Notice that if all the constructors assign a value to the final field (once), that is valid.

The last option is confusing because an array is an object. What it means is the following:

```
1. // won't work, stopped by the compiler and runtime
2. chars = new char[2];
3.
4. // will work just fine
5. chars[0] = 'C';
```

This seems weird at first, but once you understand Java, this is perfectly consistent. An array is an object. It is a reference to the array object. The

reference cannot be modified and replaced with a different value, but since an object resides in a heap at a different location, we can assign to it an arbitrary value. To make a string truly immutable, the **toCharArray()** method allocates a new array and copies the characters into it. The **record** type introduced in Java 14 also has the same limitations. All its fields are final, but they can be modified externally if they reference an object or an array. We need to take immutability with a grain of salt in Java. True immutability is expensive and sometimes not worth the cost. It might be worth practicing with discipline rather than through memory copying.

Functional programming

When reading internet posts, it would seem like functional programming is somehow contradictory to OOP. This is clickbait nonsense. Functional programming brought some mathematical ideas for expression. Most of them play very nicely with object-oriented concepts, although they do not all fit with the semantics of Java. Java is not a pure object-oriented language either. Java is a language of compromises that takes pragmatic approaches to language design. That is a "good thing™". It is a language that is not stuck in the past or stuck on some academic theory, and it is a pragmatic language for pragmatic people. That is the reason behind its decade-long popularity.

In this section, we will not dive into functional programming theory. There are other books on the subject. Instead, we will review the functional programming concepts incorporated into Java and those that are valuable in Java. Immutability is an important concept in functional programming, but it is also crucial in Java. That is why it deserved its own space.

Another important concept is idempotency. It means that when we invoke a function, we should get the same result. That seems simple enough, but it holds great power when we want to test a system. When we design an API or layers, we need to separate the calls in a way that lets us maximize idempotency so we can test the behavior consistently. If an operation returns a different result for the same input, we cannot build a test for that operation.

Java 8 introduced several functional programming concepts. The most important of them were streams. The word stream is used for two separate features in Java: one is in the context of IO to read or write from a file or network. Another case (which is the one we are discussing now) is for streams of data which mean lists or collections. The stream API lets us process elements in a functional style, which is more fluid than using the traditional for loop. Let us start with an example:

1. return numbers.stream()

```
2. .filter(num -> num % 2 == 0 && num > 0)
3. .distinct()
4. .sorted()
5. .collect(Collectors.toList());
```

numbers is a **List** containing random **Integer** objects. The **stream()** method converts the **List** to a functional interface that lets us chain independent operations. Every operation transforms the list in a small way. The **filter** method accepts a predicate interface, which is expressed as a Lambda here. The argument is the entry we are currently filtering. In this case, we are filtering out all negative and uneven numbers. The **distinct()** method removes the duplicates, and the **sorted()** method sorts the stream. The collect method converts the result back to a collection, in this case, a list. The stream code can be further refined by removing the Lambda expression from the filter and replacing it with a Java 8 method reference as such:

```
1. private static boolean test(Integer num) {
2. return num % 2 == 0 && num > 0;
3. }
4. private static Collection<Integer> evenStream(List<Integer> numbers) {
5. return numbers.stream()
6. .filter(StreamsSample::test)
7. .distinct()
8. .sorted()
9. .collect(Collectors.toList());
10. }
```

We can do the same thing using a **for** loop, as shown here:

```
1. Set<Integer> uniqueValues = new TreeSet<>();
2. for (Integer num : numbers) {
3. if (num % 2 == 0 && num > 0) {
4. uniqueValues.add(num);
5. }
6. }
7. return uniqueValues;
```

A **Set** allows only one instance per object. **TreeSet** is sorted, so we do not need to sort at the end. This only leaves the loop and numbers, which is a simple **if** statement. So which code is “better”?

Most experts argue that the stream approach is better because it expresses

the intention of the programmer. This is open to debate. There is an argument to be made for loops too. At the time of this writing, loops are typically faster for smaller and simpler cases, whereas streams have an advantage for huge sizes. With the previous code processing 100,000 numbers on Mac, it took 20 milliseconds for the stream and 14 milliseconds for the loop. Raising that number to 10,000,000 changed the picture; streams take 1,352 ms, and the loop 2,159 ms. The performance story is not clear in this case and is continuously evolving.

Conclusion

In the chapter, we reviewed the concepts that make object-oriented programming such a powerful abstraction. We reviewed some of the nuances in object-oriented design.

We dove into common design patterns and learned about their usage in the Java API. We discussed high-level functional programming concepts and how they fit in with Java's object-oriented model. We discussed immutability and its leaky nature in the world of Java objects.

In the upcoming chapter, we will review the changes that occurred between Java 8 and Java 20 and possibly further. We will discuss revolutionary concepts in the world of Java, such as Valhalla, Loom, and GraalVM.

Points to remember

- It is best to code to an interface instead of a class. However, this can be taken too far.
- Inheritance is indeed inflexible, but composition is not always the right substitute.
- Patterns are a powerful tool for communication and for building a mental model of sophisticated systems.
- Java 8 added powerful concepts from functional programming, most importantly streams. Streams let us replace for loops with a more fluent syntax.
- Immutability and idempotency are powerful tools when designing a system, and we should strive for more of both.

Multiple choice questions

A. What is double-checked locking?

1. When we have two locks
2. When looping over a collection with an iterator and accessing the list

again

3. When a field is declared final but we can still mutate its variables

4. When we check a condition, then check it again within a synchronized block

B. Which two patterns are similar?

1. Singleton
2. Façade
3. Proxy
4. Iterator
5. Factory

C. Which of the following Java APIs is immutable?

1. String
2. Integer
3. Byte
4. Boolean
5. Short
6. All of the above

D. ConcurrentModificationException can happen when iterating because…

1. We modified the underlying collection
2. A different thread modified the collection
3. We used an iterator instead of a for loop
4. We have two concurrent iterators
5. All of the above

Answers

A. 4

B. 2 and 3

C. 6

1 https://www.gofpatterns.com/

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 3
8 to 21 to GraalVM

Introduction

Java went through many pivotal releases. Java 1.1 introduced inner classes. Java 2 introduced Swing and collections. Java 5 introduced Generics. However, one of the most pivotal releases was Java 8, which introduced a slew of wide-sweeping changes. Up until Java 8, the releases of Java versions were irregular, although they tended to follow a two-year cadence. With the release of Java 9, this changed. We now have a six-month release cycle. This pace made the period between Java 8 and Java 21 relatively short.

Another major change introduced was the ability to preview features. Java ships with features in preview mode. They are off by default, and you need to enable them explicitly to try them. These preview features might break or might be removed altogether. They are there for experimentation and feedback only. This has been a valuable feature for Java's stewards, who can try their changes in the real world and decide what works. This provided the right environment for language growth. As a result, Java has added many features in the last decade.

We will start by talking about the Java 8 release and what it brought to the table. This is important to form a baseline for all future discussions. The rest of the chapter is divided into the following three types of changes:

1. **VM changes**: Changes to the core virtual machine that might not impact the language but make a big difference.
2. **Language changes**: Syntax sugar and Java language refinement.
3. **APIs**: New APIs that simplify and enable new capabilities.

Finally, we will discuss GraalVM, which is an exciting virtual machine. For brevity, we will skip features that were later removed from the JDK. This

means that we will not discuss JavaFX, JAXB, Pack200, and so on. Since this is a book about the language, we will also skip changes to external tools unless they are of particular importance. We will also skip small changes and focus on the big-ticket items that are more likely to be noticeable.

Structure

In this chapter, we will discuss the following topics:

- Java 8—the baseline
  - Lambda expressions
  - Method references
  - Default methods
  - Streams
  - Annotation changes
  - Method parameter reflection
  - New date and time API
- VM changes
  - Modules (Jigsaw) (Java 9)
  - Shenandoah (Java 12)
  - Microbenchmark Suite (Java 12)
  - Virtual threads—Loom (Java 21)
  - Deprecation of finalization (Java 9)
  - UTF-8 by default (Java 18)
- Language changes
  - Try with resources
  - Private methods in interfaces (Java 9)
  - var Keyword (Java 10)
  - Switch expression (Java 14)
  - Sealed classes (Java 17)
  - Pattern matching instanceof (Java 16)
  - Text blocks (Java 15)
  - Records (Java 16)
  - Record patterns (Java 19 Preview)
  - String templates (Java 21 Preview)
  - Unnamed patterns and variables (Java 21 preview)
- APIs

    - HttpClient (Java 11)
    - Foreign function and memory API—Panama (Java 19)
    - Structured concurrency (Java 19)
    - Serialization filtering (Java 9)
    - Scoped values (Java 20)
    - Sequenced collections (Java 21)
- Future
    - GraalVM
    - Valhalla
    - Vector API

Objectives

By the end of this chapter, you will understand what modern Java has to offer and why these features were added to Java. You will be up to date with the latest and greatest capabilities of the JVM.

Java 8—the baseline

Java 8 was released in March 2014. It was the first **Long Term Support** (**LTS**) Release under the new release cadence. This was the second Java release since Oracle acquired Sun (finalized in January 2010), but it was the first release developed under Oracle's stewardship. Oracle had a lot to prove. It had to show the Java community that it was serious about moving Java forward. They delivered; Java 8 is still considered one of the most pivotal releases in Java's history.

Lambda expressions

We discussed Lambda expressions before, but they are worth revisiting since they are such an important feature. A Lambda is an encapsulated block of code that can act as a callback. The following code shows the creation of a thread before and after Java 8:

```
// Pre-Java 8
Thread t = new Thread(new Runnable() {
public void run() {
// do this on other thread
}
});

// Post Java 8 using Lambda
```

```
9. Thread t = new Thread(() -> {
10. // do this on other thread
11. });
```

A major benefit of Lambda is the shorter-fluent syntax. Reduction of noise is indeed a major benefit of this feature. But there is another subtle difference and an advantage. The difference is in the scoping behavior. An anonymous inner class has its own scope, but a Lambda is considered a part of the surrounding scope. Since the anonymous inner class can access variables in the parent scope, the difference is very subtle, but it is there.

Take into consideration the following code:

```
1. Runnable r = new Runnable() {
2. public void run() {
3. field.invokeMethod();
4. }
5. };
6.
7. Runnable r = () -> {
8. field.invokeMethod();
9. });
```

Let us say the field is **null** when we create the instance of the **Runnable** object. After we create the object, we set the field to a new value. The inner class will still have a null value, whereas the Lambda will not. Why is that?

An inner class (anonymous or otherwise) is a separate class. The compiler adds a hidden constructor that passes the extra parameters, and they are copied on construction. This can create subtle odd behaviors, such as a **null** reference to **this** in some edge cases. But the important thing to keep in mind is that values are copied in construction. A Lambda will reference the actual variables and will update as the variables update.

This has a second implication. When we compile Java code, it is compiled into a format called bytecode. This format is a relatively simple machine code format that matches the source code relatively closely. The JVM compiles or interprets that bytecode when running the app. A Lambda expression does not generate the same bytecode as the new expression you see above it. While it will act in a very similar way, it is still a different implementation under the hood. A Lambda will use the **invokedynamic** bytecode, which is a new bytecode instruction added in Java 7. This means the Java 8 block will not allocate a new **Runnable** object with every

execution of that block.

A Lambda is more beautiful and faster.

Method references

Lambdas are great, but they can end up as long and distracting blocks of code. We might want to extract some lambda logic to a method for reuse on other blocks. Or we might want to invoke just one method; why do we need a lambda for that?

Method references are a different syntax for Lambdas that invoke a single method. This is not 100% accurate because there are some subtle differences in implementation. But semantically, they are very close. We can convert any Lambda to a method reference and vice versa. Continuing the thread example from the Lambda explanation, this would be the method reference version of that:

```
1. // Call a method from the thread
2. Thread t = new Thread(obj::runThisMethodOnAnotherThread);
```

Notice that we do not pass brackets to the method reference. It is assumed that the method does not accept arguments here. But this is not always the case. If the Lambda accepted arguments, the method would need to have an identical signature to process the argument.

Default methods

Up until Java 8 interfaces could only have abstract methods. This created a problem; let us say you want to add a method to an interface, but some person has already implemented that interface. Their code will no longer compile. Surprisingly, it would run and work if you did not invoke the missing method. But that was inconvenient and created a disincentive to user interfaces. Developers ended up using single-method interfaces and creating multiple separate interfaces just to avoid that problem.

Java 8 solved this by introducing default methods. We can add non-abstract methods to an interface. Such methods do not do much; they cannot modify fields (fields are all final), but it is a powerful tool, as you can see in the following code:

```
1. public interface Greeting {
2. void hello();
3.
4. static Greeting createGreeting() {
5. return new GreetingImplementation();
```

```
}

default void goodbye() {
System.out.println("Goodbye");
}
}
```

Notice that static methods in interfaces were also included in this change and are now legal.

Streams

One of the big drivers for the introduction of the previous three features is the streams API. We discussed this in *Chapter 2, OOP Patterns*, so we will not go into too many details about it. Just remember that streams would not have been possible without Lambdas and method references. The syntax would have been unreadable.

Streams would not have been practical without default methods because it required changes to the collection API interfaces.

Annotation changes

Java 8 introduced a couple of changes to the annotation system introduced in Java 5. The first is the ability to repeat an annotation. This lets us declare the same annotation more than once, for example:

```
@Allow(group = "Managers")
@Allow(group = "SeniorManagers")
public void managementConsole() {
// ....
}
```

Another capability is the “type annotation”, which is the ability to place an annotation where a type is declared. Check the following code:

```
public @NotNull ResponseObject method(@NotNull String arg) {
// ...
}
```

This lets us declare that the argument and the response will not be null. That is great for us, but it is even more important if our code should interact with Kotlin code. Kotlin can declare variables that cannot be null. By declaring nullability with such an annotation, we can improve the interaction between Java and Kotlin.

Method parameters reflection

We discussed reflection in *Chapter 2, OOP Patterns*. One of the big features of reflection is discovery. The ability to find out which methods, fields, and so on are a part of the class. We can query the methods using the following code. This is not new functionality. What is new is the ability to get the names of the arguments:

```
Method[] methods = Object.class.getMethods();
for(Method m : methods) {
System.out.print(m.getName() + "(");
for(Parameter p : m.getParameters()) {
System.out.print(p.getName() + " ");
}
System.out.println(")");
}
```

New date and time APIs

The Calendar and Date classes in Java are notoriously difficult to work with. Java 8 introduced new APIs inspired by community projects (specifically, Joda Time) as part of **Java Specification Request** (**JSR**) 310. This API includes many new capabilities, but more importantly, it is better structured as an API. For example, to get the current date, we would do something like the following:

```
// Prior to Java 8
Date currentDate = new Date();

// Java 8 API:
LocalDate localDate = LocalDate.now();
```

At first glance, these might seem equivalent. The old approach might even seem shorter or simpler. But there is a basic subtle improvement here. The old date object defaults to the current date. That is not clear from the constructor. Using the factory method in the local date API is very clear. The API is also overloaded with capabilities that let us get the current time in a specific time zone, parse a date, and so much more. Refer to the following example:

```
LocalDate.of(2023, 01, 1);
LocalDate.parse("2023-01-01");
```

A great example is date arithmetic. Calculating a future or past date is

difficult; let us say we want to find a date five weeks in the future. We can do the following:

```
// Prior to Java 8
calendar.add(Calendar.DAY_OF_YEAR, 35);

// Java 8 version
LocalDate fiveWeeksFromTime = myLocalDate.plus(5, ChronoUnit.WEEKS);
```

Again, the old version seems shorter. But the flaws are much greater. Notice that the API is not well-typed. The add method accepts any argument, and mistakes are easy to make. However, the biggest problem is mutability. The API changes the state of the calendar object. It is more error-prone as a result.

The Java 8 version is clearer and much simpler. It is immutable, which might have some performance impact. But that is probably worth the cost for the immutability.

This is a tremendous API filled with nuances. We will revisit it in the rest of the book.

VM changes

There have been many changes to the Java Virtual Machine since JDK 8. These are some of the big highlights currently available or in preview.

Modules AKA Jigsaw (Java 9)

Jigsaw is possibly the biggest change to the JVM and API since the launch of Java. Up until Java 9, we could associate classes with one another and within a package. Jigsaw provided an additional layer of isolation: a module. Think of a module as an additional level of encapsulation and isolation.

Packages might have inter-dependencies and internal implementation details that we might not want to expose when publishing an API. With modules, we can export only the applicable pieces. Furthermore, we can pack only the applicable pieces that matter. The JDK itself is modularized, and you can define specific module dependencies. If you do not need the other modules, you can create a custom JVM without the unnecessary modules. This helps us think in boundaries and clearly define our dependencies.

To get started with Jigsaw, we need to add a module definition file. This is a file named **module-info.java**; notice that this is an illegal class name in

Java due to the—character. That means there is no chance of a name collision with a class. We place that file in the top package of the module hierarchy. For example, say we have the packages:

- **com.debugagent.myapp.ui**
- **com.debugagent.myapp.model**
- **com.debugagent.myapp.controller**

Then the **module-info.java** file will reside in **com.debugagent.myapp**. We can define the module as follows:

```
1. module com.debugagent.myapp {
2. requires java.logging;
3. exports com.debugagent.myapp.model;
4. }
```

There are a couple of interesting things to notice here. We define requirements for a different module (a dependency). We also define the packages that this module exports. That means external projects using this module will only see the model package and will not see the UI or controller packages.

Here is an interesting and hard-to-understand concept. Let us say we have three modules: A, B, and C.

Module A defines a Class A. Module B requires Module A and defines a method with the signature **public AClass getMyClass()**. Would Module C work if it only requires Module B?

No. The reason is that **AClass** will be hidden from Module C as the definition is in Module A. The solution here is to use the **transitive** keyword. It makes the required keyword implicitly work for all other modules. It means that anyone who includes Module B implicitly includes Module A. We need that if we want to expose classes that are defined in Module A.

This sounds like a lot of pain, another layer of encapsulation. You would not be wrong to think that. The adoption of Jigsaw is ongoing and slow. However, newer frameworks are leveraging some of the advantages brought by the module system. GraalVM can compile faster, Spring handles dependencies better, and the JVM can be shrunk, which can matter for cloud deployments. There are also additional suggestions that can significantly shift the balance with regard to modules, such as: **https://openjdk.org/jeps/8305968.**

Shenandoah (Java 12)

The JVM has several **Garbage Collectors** (**GCs**) built in. Shenandoah is the latest one. GCs are amazing, unreferenced objects that are collected in an automatic process. However, garbage collection still makes some tradeoffs:

- Memory footprint
- Pauses
- Performance

Pick two of those. A garbage collector can tradeoff RAM to provide faster performance and fewer GC stalls. Normally, when we want to pick a library, we just do a benchmark. But benchmarking a GC is much harder. If we overload a GC, we might end up with a GC that handles stress well but is not optimal for typical memory allocation. It is crucial to understand how the garbage collectors work and that we profile the GC with "real world" workloads.

Not your fathers stop the world mark sweep

Java GCs have come a long way since Java 1.0's stopped the world GC. While there are many types of garbage collectors, most of the new ones are generational and parallel/concurrent. This might not seem important when working on our local machines. But the difference is very noticeable when GCing is very large heaps.

GCs "seamlessly" detects unused objects to reclaim heap space. But there are tradeoffs.

Generational garbage collection

Most modern GCs assume the object life-cycle fits into a generational paradigm. The old-generation space objects live a long life and rarely get collected. They do not need frequent scanning. Younger generations of objects live and die quickly. Often together.

Generational garbage collection (typically) traverses the young generation more frequently and gives special attention to connections between the generations. This is important as there are fewer areas to scan during a minor garbage collection cycle. The term for the shorter cycle is incremental GC as opposed to the full GC cycle. GCs typically try to minimize full GC cycles.

Concurrent versus parallel garbage collector

Parallel GC is often confused with concurrent GC. To make matters even more confusing, a GC can be both a parallel GC and a concurrent GC (for example, G1).

The difference is simple, though:

- A parallel GC has multiple GC threads. The GC threads perform the actual garbage reclaiming. They are crucial for large-scale collection.
- A concurrent GC lets the JVM do other things while it is in the mark phase and optionally during other stages.

Intuitively, most of us would want to have both all the time and make use of application threads. But this is not always the right choice. Concurrency and multiple application threads incur an overhead. Furthermore, these GC's often make tradeoffs that miss some unreachable objects and leave some heap memory unreclaimed for a longer period. To be clear, they will find all unused memory in a full GC cycle, but they try to avoid such cycles, and you might pay the penalty.

Here are the current big-ticket collectors prior to Shenandoah. They are still great GCs that can provide optimal performance under the right conditions. We need to experiment with each one of them to understand which one best suits our needs.

Serial collector

This is a single-thread garbage collector. That means that it is a bit faster than most GCs but results in more pauses. If you are benchmarking performance, it makes sense to turn on this GC to reduce variance. Since pretty much every CPU is multi-core, this GC is not useful for most real-world deployments, but it makes debugging some behaviors much easier.

There is one case where serial collectors might have a big benefit in production, and that is serverless workloads (for example, lambdas and so on). In those cases, the smallest/fastest solution wins, and this might be the right solution to work with limited physical memory on a single-core VM.

Notice that despite its relative simplicity, the serial collector is a generational GC. As such, it is far more modern than the old Java GCs.

You can turn on this GC explicitly using the **-XX:+UseSerialGC**.

Parallel collector or throughput collector

The multi-thread equivalent of the serial collector. This is a fine collector that can be used in production, but there are better choices in some cases. The serial collector is better for benchmarks, and ZGC/G1 usually offers better performance.

You can turn on this GC explicitly using the **-XX:+UseParallelGC** option.

One of the big benefits of parallel GC is its configurability. You can use the following JVM Options to tune it:

- **-XX:ParallelGCThreads = ThreadCount**: The number of GC threads used by the collector.
- **-XX:MaxGCPauseMillis = MaxDurationMilliseconds**: Place a limit on GC pauses in milliseconds. This defaults to no limit.
- **-XX:GCTimeRatio = ratio**: Sets the time dedicated to GC in a 1/(ratio + 1), and so, a value of 9 would mean 1/(9 + 1) or 10%. So, 10% of CPU time would be spent on GC. The default value is 99, which means 1%.

G1 Garbage Collector

The G1 Garbage Collector is a heavy-duty GC designed for big workloads on machines with large heap sizes (roughly 6GB or higher). It tries to adapt to the working conditions in the given machine. You can explicitly enable it using the JVM option **-XX:+UseG1GC**.

G1 is a concurrent GC that works in the background and minimizes pauses. One of its cooler features is string de-duplication which reduces the overhead of strings in RAM. You can activate that feature using **-XX:+UseStringDeduplication**.

Z Garbage Collector (ZGC)

ZGC is designed for even larger heap sizes than G1 and is also a concurrent GC. It does support smaller environments and can be used for heap sizes as small as 8 MB all the way up to 16 TB maximum heap size!

One of its biggest features is that it does not pause the execution of the application for more than 10 ms. The cost is a reduction in throughput.

ZGC can be enabled using the **-XX:+UseZGC** JVM option.

Shenandoah

The Shenandoah GC depends a lot on heap size. It performs best when there is enough space for all the allocations when the GC is running concurrently. It works well with up to 128 GB of heap size and can go as low as 1 GB. It is designed for a larger scale, similar to G1 and ZGC. Shenandoah has longer GC pauses than ZGC (10 ms versus 1 ms); it has a high tail latency effect compared to ZGC, which means you can see stalls.

This is a tradeoff where we will pay in longer pauses but get overall better performance when compared to ZGC. If consistent performance matters more, then ZGC is probably superior. If raw performance and overall

throughput are the determining factors, then Shenandoah might be the better option.

Note that not all JDK distributions include Shenandoah, so it might not be available. When it is available, it can be activated with the **-XX:+UseShenandoahGC**. All of the GCs mentioned here have many flags and fine-tuning options, but that is outside the scope of this book.

Microbenchmark Suite (Java 12)

The microbenchmark suite was added as a part of the JDK **Enhancement Proposal** (**JEP**) 230. The motivation behind this is rather elaborate. When we want to test if something performs well, we usually just write a simple test that runs it in a loop and measures the time it takes. This works for Java but is very inaccurate. Java programs are usually executed with a **Just in Time** (**JIT**) compiler. Code that runs in a JIT can start slow and then get recompiled for faster performance as it executes. To make matters even worse, a GC can have an uneven impact on performance that can completely skew test results.

These problems led to many falsehoods and mistakes in the world of Java. There was no single, unified way to test the performance of a method. Brian Goetz (Java Language architect) eloquently described the problem: "The scary thing about microbenchmarks is that they always produce a number, even if that number is meaningless. They measure something, we are just not sure what."

To use the microbenchmark framework, we need to configure Maven a bit differently than we normally do. We need to create a maven archetype based on JMH to include the necessary annotations, as shown in *figure 3.1*.

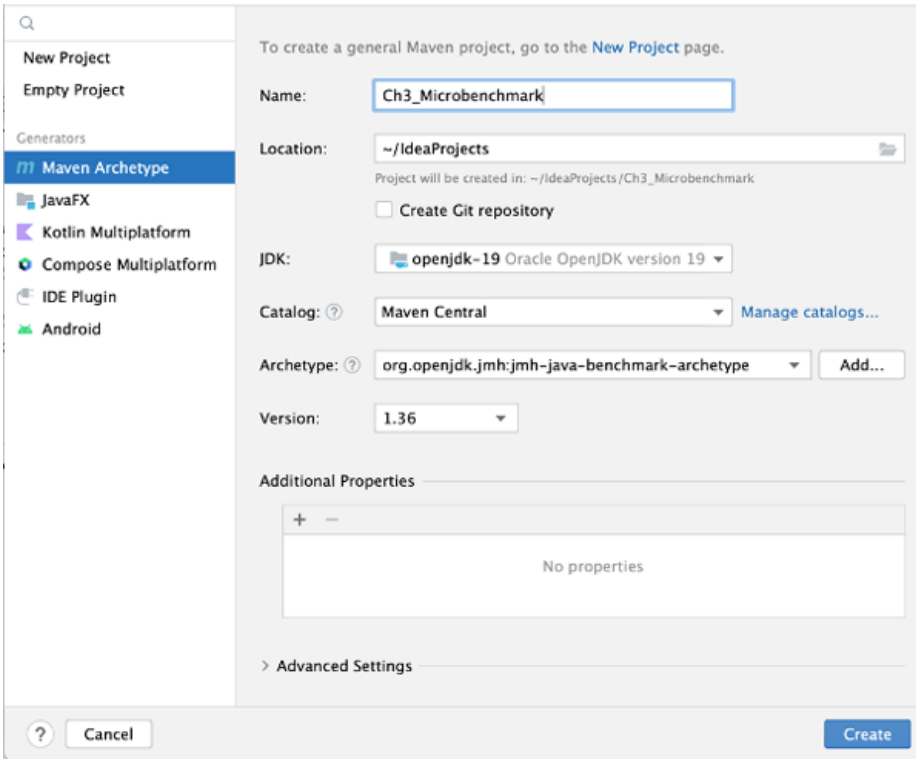

Figure 3.1: *Creating a Maven project archetype for JMH in IntelliJ/IDEA*

Once that is created, we end up with a skeleton project that includes a simple benchmark sample. You can find the full code on GitHub[1]. The benchmark itself works with Java methods and annotations, as shown in the following code:

```
1. public class MyBenchmark {
2. private static final int[] ARRAY = new int[1000];
3. private static final List<Integer> LIST = new ArrayList<>();
4.
5. static {
6. Random random = new Random();
7. for(int iter = 0 ; iter < ARRAY.length ; iter++) {
8. ARRAY[iter] = random.nextInt();
9. LIST.add(ARRAY[iter]);
10. }
11. }
12.
13. @Benchmark
```

```
14. public long primitiveArrayPerformance() {
15. var result = 0L;
16. for(var current : ARRAY) {
17. result += current;
18. }
19. return result;
20. }
21.
22. @Benchmark
23. public long listPerformance() {
24. var result = 0L;
25. for(var current : LIST) {
26. result += current;
27. }
28. return result;
29. }
30. }
```

The benchmark methods are annotated and compare the difference in performance between iterating over a list and over an array. Notice that we return the result of the sum. This is important; otherwise, the compiler might conclude that the code does nothing and eliminate it. Running this produces a lot of output as the benchmark runs five cycles of a warmup followed by five cycles of tests. It then produces a report highlighting the performance differences between the two. There is a lot of output above this, but the bottom line is, as shown in the following output of the command:

```
Benchmark                                 Mode  Cnt        Score       Error
Units

MyBenchmark.listPerformance              thrpt   25  1654871.542 ± 11403.120
ops/s

MyBenchmark.primitiveArrayPerformance  thrpt   25  1772287.412 ± 12783.236
ops/s
```

This is in Java 19, prior to features such as Valhalla, and primitive arrays are not much faster than lists. Historically the difference was in the 20× scale. The improvement is tremendous.

Virtual Threads—Loom (Java 21)

One of the best changes to Java after version 8 would probably be virtual

threads. To understand virtual threads, we first need to understand a bit of history and the motivation behind this. When Java was first launched, we had green threads. Threads were implemented entirely in Java, and the virtual machine itself appeared as a single thread to the OS.

This was problematic. If one thread was stuck, the entire JVM was stuck. Worse, we could not leverage native capabilities on separate OS threads. As a result, Java switched to using native threads, a system it uses to this day. Native threads are great. They are fast, and they use hardware resources. We can use multiple CPU cores effectively by having a different thread on every core; it is almost seamless to use as developers.

The Java API embraced threads. Every IO read operation, and every server call runs on its own thread. This worked great in the age of monoliths, but it became a problem. With the rise of microservices, we built many small applications running on weaker containers. A container is judged by the number of connections it can handle. The more it can handle per-request, the better.

Unfortunately, Java peeked at a low number for throughput. It could not handle many network requests because, in the world of Java, every request required a thread. Other platforms that used a single-thread approach performed better, which is surprising but makes sense when you understand the problem. A Web server spends most of its time idle. Even when under heavy load, it is always waiting for IO; for a request from a browser or a request over the network to the database or a Web service. In Java, threads would spend all their time blocking reading a request from a client or blocking reading data from a database. Either way, the threads were just sitting there doing nothing.

This begs the question, why do we not add more threads?

OS Threads are very expensive in large numbers. The OS does not know in advance what you plan to do with the thread and needs to allocate a lot of resources to the thread. It then has a lot of overhead switching between threads since it does not know what the thread did, and so it needs to treat it as if it changed everything. Once you cross the hundreds of threads barrier, performance plummets. That is why Java developers usually use thread pools. To reuse existing OS threads and avoid the creation of new ones.

One approach is asynchronous APIs which Java frameworks have added. These were able to work around the problem and provide the level of throughput other languages enjoyed. But Javas syntax was designed for synchronous coding. Some languages added keywords such as async/await to make asynchronous code more readable. But Java chose a different

path. To understand why, let us first explain why asynchronous code is problematic even when we have async/await.

With an asynchronous process, we can read the IO differently; when it blocks, the event loop just passes the buck to a new task. The IO process will send a callback when more data is ready. This works well. However, it means that stacks are disconnected. When you have a failure locally or in production, you do not have a context but only the current variables and data. There is no chain.

This gets worse. Imagine a single transaction that is atomic. All changes need to go in or they should all fail. This is a basic concept of databases. But if everything is asynchronous, it is much harder to perform complex tasks atomically. Synchronous programming is faster, simpler, easier to debug, and better in every way. Except throughput. It does not scale nicely.

Loom

Project Loom is the codename for virtual threads. The concept is simple: instead of using native threads, a virtual thread is like a green thread. But unlike a green thread, it can use multiple native OS threads under the hood (as many as we have CPU cores). This means it enjoys the ability of native threads to use the hardware to the fullest while reducing the per thread overhead to nothing.

As a point of comparison, a native thread can have a 2 MB RAM overhead, whereas a virtual thread has around 1 kB. This makes virtual threads effectively free. Creating a million threads is possible and affordable.

To make matters even more enticing, Loom requires almost no code changes. Existing code is oblivious to virtual threads. You do not need to adapt it like you would with an asynchronous framework. There are no downsides to using virtual threads, but there are limitations.

If we are writing CPU-intensive code, it might make more sense in a native thread. Otherwise, some of the other virtual threads might be impacted. You can mix virtual threads and native threads by using the existing thread creation APIs. Notice that you should never pool virtual threads, as it makes no sense. At the moment, Loom is still in preview release, but it is very stable and efficient. To use experimental features in IntelliJ/IDEA, we need to perform several steps. First, open the module settings (right-click the menu on the project) and select the **experimental feature**, as seen in *figure 3.2*.

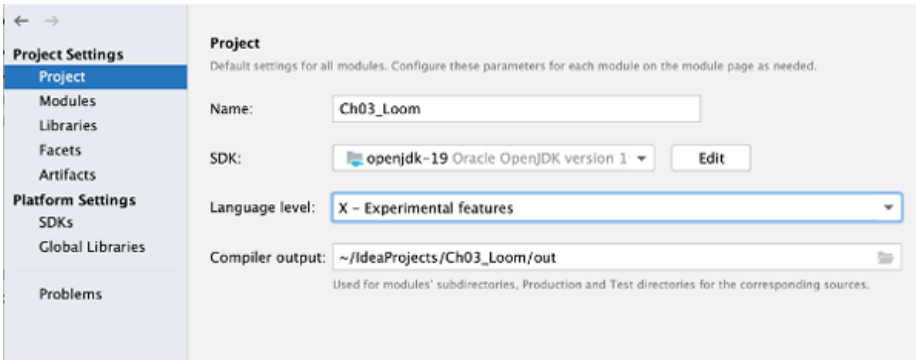

Figure 3.2: *Experimental features in IntelliJ/IDEA*

Next, we need to go to the settings or preferences area (depending on your OS) and select **Java Compiler** section (you can use the search to find it). We need to add the **--enable-preview** option, as seen in *figure 3.3*.

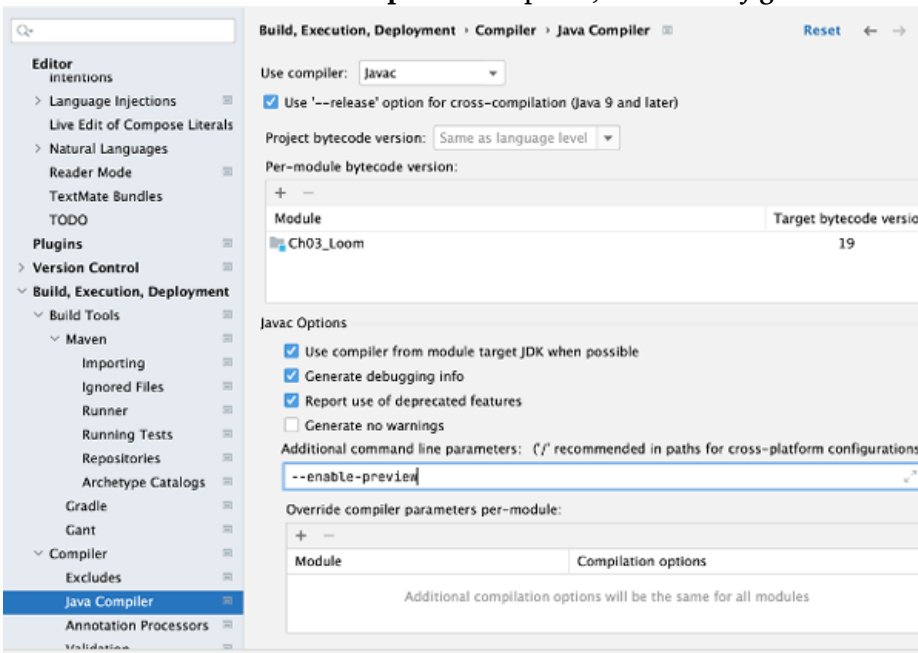

Figure 3.3: *Adding --enable-preview to the Compiler options in IntelliJ/IDEA*

Finally, we need to add **--enable-preview** to the run configuration VM arguments, as seen in *figure 3.4*.

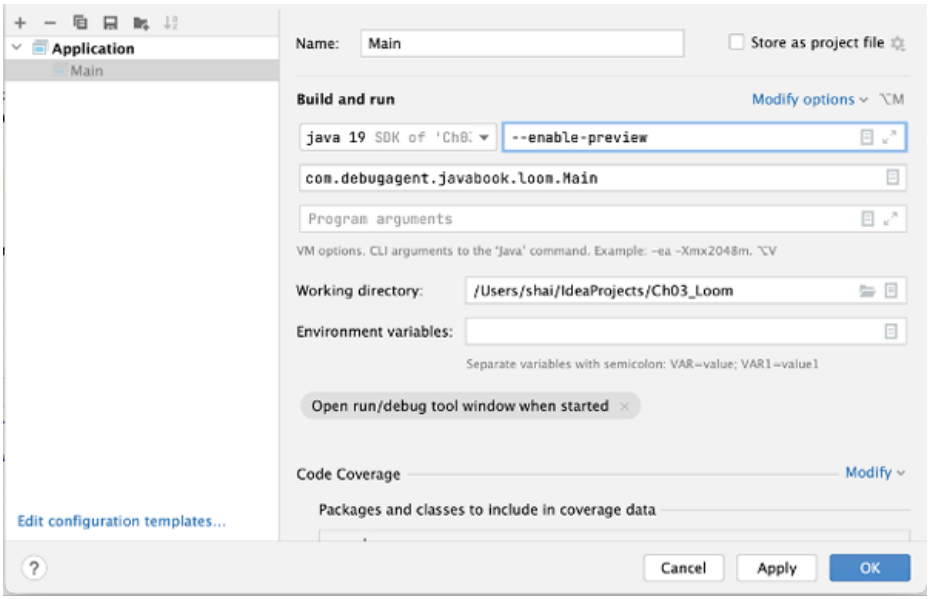

Figure 3.4: *Adding --enable-preview to the run configuration VM options in IntelliJ/IDEA*

Notice that the VM arguments are hidden by default, and we should show them explicitly via the modify options menu, as seen in *figure 3.5*.

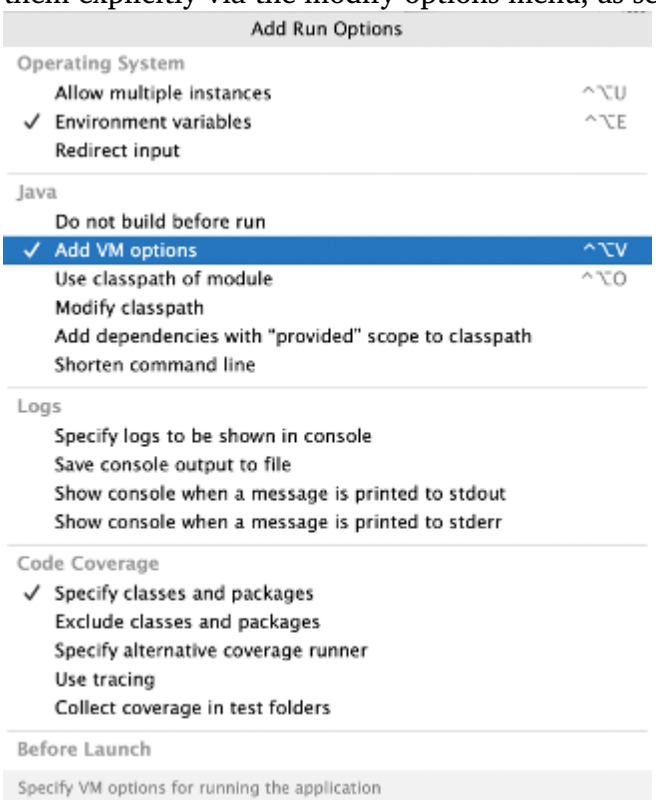
Add Run Options
Operating System
Allow multiple instances ^⌥U
✓ Environment variables ^⌥E
Redirect input
Java
Do not build before run
✓ Add VM options ^⌥V
Use classpath of module ^⌥O
Modify classpath
Add dependencies with "provided" scope to classpath
Shorten command line
Logs
Specify logs to be shown in console
Save console output to file
Show console when a message is printed to stdout
Show console when a message is printed to stderr
Code Coverage
✓ Specify classes and packages
Exclude classes and packages
Specify alternative coverage runner
Use tracing
Collect coverage in test folders
Before Launch
Specify VM options for running the application

Figure 3.5: Enabling the VM options text box in IntelliJ/IDEA

Once this is done, we can write code that relies on preview and experimental features such as Loom. With that, we can use the following code to create multiple virtual threads:

```
1. AtomicInteger atomicInteger = new AtomicInteger(0);
2. for(int iter = 0 ; iter < 1000 ; iter++) {
3. Thread.startVirtualThread(() -> System.out.println("Loom thread " +
4. atomicInteger.addAndGet(1))
5. );
6. }
```

This is basic stuff. The impact of Loom is felt when building server applications that need to handle scale.

Deprecation of finalization (Java 9)

Java 9 deprecated the concept of finalization. Various versions deprecated and removed many features; why is this feature so important that we are giving it a platform here?

To understand this, we need to understand finalize. Java does not have destructors. It does not have a method that is invoked when an object is destroyed. It does have a **finalize()** method, which is similar. It is invoked by the GC to indicate that the object will be destroyed. Why do we need that?

Let us say your Java code maps to OS native code, for example, a file object or a network connection. If the GC removes the object, we would want to also remove the underlying native resource to clean up the memory. That is a noble goal. But it also triggers many problems; for example, the GC needs to invoke the finalizer. That means as it “wants” to collect an object: there is a finalizer, so it cannot collect the object, and it needs to move it to a different collection process (which is slow).

The finalizer itself is invoked on a GC internal thread and can cause issues or race problems. The finalized method can change the application state and even create a reference back to the object, essentially pulling it back from GC. This is a security problem and a technical problem with a deep performance impact. So, good riddance.

Java 9 added the **Cleaner** API, which lets us do something similar without the problems of finalization.

UTF-8 by default (Java 18)

Character encoding is one of the more painful subjects to understand. One of Java's biggest innovations was picking up Unicode as the default. It was a remarkable leap forward. However, when reading text files, it could create strings from bytes or pretty much any operation. Java defaulted to using the system encoding. Encoding is the process of deciding how bytes convert to characters. Each OS in every locale has its own encoding, and back in the 90s, UTF-8 was not common.

In the past decade, UTF-8 took over and became the de-facto standard encoding. It solves many of the problems that plague character conversions. By using it as the default, code can become much simpler, as shown in the following code:

```
// Prior to Java 18, this method throws a checked exception!
var str = new String(bytes, "UTF-8");

// Post Java 18 this will still be UTF-8
var str = new String(bytes);
```

Language changes

Many language changes have come out since JDK 8, but none of them were as big as Lambdas. This might arguably change once Valhalla ships, but that is not even at the preview stage at this point. We will discuss Valhalla in the future section.

Try with resources

Closing a file is a pain. Especially since **close()** itself can throw an exception. This small change makes a huge difference, as you can see from the following code:

```
// Using try with resources
try(var input = new FileInputStream("pom.xml")) {
System.out.println(new String(input.readAllBytes()));
}

// Old way
InputStream input = null;
try {
input = new FileInputStream("pom.xml");
System.out.println(new String(input.readAllBytes()));
} finally {
```

```
12. try {
13. if(input != null) {
14. input.close();
15. }
16. } catch (IOException exception) {}
17. }
```

In a new way, we can allocate the closable element directly in the try brackets. We can include multiple closable elements and separate them with semicolons. We can also have a catch clause, which, in our case, we have left out and just let the exception propagate.

In the second block, notice that we must define the input outside since the constructor may throw an exception. Since close can also throw an exception, we need to handle that separately. This is not an ideal block of code as there might be significance to the error message on the close operation, but we chose to keep the code concise. Any object that implements the **AutoCloseable** interface can be used for try-with-resources.

Private methods in interfaces (Java 9)

Starting with Java 9, we can add private methods to interfaces. Static or otherwise. This makes sense if you have static or default public methods. In that case, you can reuse code between several such methods.

var keyword (Java 10)

We have already used the **var** keyword quite a few times. It creates an implicit definition of the type. Starting with Java 11, it can be used as arguments to Lambda callbacks as well, so we can cut down on some of the definition boilerplates.

The **var** keyword cannot be used for fields and cannot be used in cases where ambiguity exists. It only makes sense when the compiler can determine the type. It can work with primitives and will serve as a primitive when used in that way.

Due to the way **var** works, it must always be assigned at the point of creation. We cannot declare a variable with **var** without assigning a value.

Switch expression (Java 14)

Switch is a bit of a problematic statement. It is often used with an enum, but in most cases, the “right thing” would be to add a method into the enum. However, a switch in Java can work with enums that are not

controlled by the developer, with numbers, strings, and objects. It is a very powerful tool that can save us from complex structures and nested if statements.

Unfortunately, the standard version of the switch in Java is problematic. See the following example:

```
1. switch (value) {
2. case A:
3. result = "The A Value";
4. break;
5. case B:
6. result = "It's B";
7. break;
8. case C:
9. result = "It's C";
10. break;
11. }
```

There are several problematic things here. Some are subtle, but some are obvious, given as follows:

- The code is very verbose.
- If we forget a break, the code will fall over to the next statement, and we will get the wrong value.
- If a D enum is added, the code will fail.

About the last part, we can add a default switch statement that throws an exception. But that is bad. It means the failure will happen at runtime, not at compile time. We can do better with switch expressions:

```
1. result = switch (value) {
2. case A -> "The A Value";
3. case B -> "It's B";
4. case C -> "It's C";
5. };
```

This solves all the problems mentioned:

- It is not verbose.
- It does not rely on breaks.
- If we add a D enum, the code will not compile. It must be exhaustive!

We can also use yield instead of break to make use of a more traditional switch statement, as shown here:

```
result = switch (value) {
case A:
System.out.println("A reached");
yield "The A Value";
case B:
System.out.println("B reached");
yield "It's B";
case C:
System.out.println("C reached");
yield "It's C";
};
```

With JDK 19, we have a new type of pattern-matching switch expression. Say we have a shape object and would like to create custom code for various subclasses. It is in a third-party library, so we cannot do that via inheritance, which would normally be the preferred way. The normal way of solving this would be a series of **if** statements and **instanceof** tests. The pattern-matching switch simplifies the code as follows:

```
var height = switch (myShape) {
case Rectangle r -> r.height;
case GeneralPath c -> c.getBounds().height;
default ->
throw new IllegalArgumentException("Unrecognized shape");
};
```

Notice the code removes the need for a cast. We already have a variable of the right type. Also, notice that the default statement is required; otherwise, the switch statement would not be exhaustive. The compiler will not allow it.

There is a solution for that in sealed classes, which we will explore next.

Sealed classes (Java 17)

Sealed classes make sense for an API provider that wants to guard the hierarchy against modification. Java provides the **final** keyword for that. The string is final, which means that we cannot extend it. That is not bad. But what if Oracle wants to extend String?

They cannot, either. Well, they can, but they will have to make String non-final, and that would mean anyone can extend it. That is a problem and a potential security risk. What we sometimes want is a class that is final to a limited group of subclasses that we know about. That is a sealed class. The syntax looks like the following:

```
public sealed class SealedBase permits SealedA, SealedB {
}
public final class SealedA extends SealedBase {
}
public final class SealedB extends SealedBase {
}
```

Notice that **SealedA** and **SealedB** must be **final** or **sealed**; otherwise, the code will not compile. Once we have that hierarchy, we can write a switch without a default clause since the compiler knows there are no additional options:

```
switch (base) {
case SealedA a -> out.println("It's A");
case SealedB b -> out.println("It's B");
case SealedBase theBase -> out.println("It's The Base");
}
```

Pattern-matching instanceof (Java 16)

The pattern-matching **instanceof** is one of the simplest changes here. Just a small bit of syntax sugar that makes the code so much better and needs very little explanation:

```
// Before Pattern Matching instanceof
if(number instanceof Float) {
Float f = (Float) number;
return Math.round(f);
}

// After pattern matching
if(number instanceof Float f) {
return Math.round(f);
}
```

Since we know the type after the **instanceof** statement, it is very likely that we will want a variable with that type. We just add the pre-assigned

variable to the statement, and it gets created in the scope. Perfect and simple.

Text blocks (Java 15)

Developers who come to Java from text processing languages often find the strings painful. Java inherited a lot of its String syntax and sensibilities from C, and the result is verbose. Especially when we need to represent a large block of text. This is where text blocks solve the problem:

```
1. public static String regularStrings(String title, String link) {
2. return
3. "<html>\n" +
4. "<head>\n" +
5. "<title>" + title + "</title>\n" +
6. "</head>" +
7. "<body>\n" +
8. " <a href=\"https://debugagent.com\">" + link + "</a>\n" +
9. "</body>";
10. }
11.
12. public static String textBlockString(String title, String link) {
13. return
14. """
15. <html>
16. <head>
17. <title>%s</title>
18. </head> <body>
19. <a href="https://debugagent.com">%s</a>
20. </body>
21. """.formatted(title, link);
22. }
```

The classic string arithmetic is painful to write. We need to open and close quotes all the time. Escape quotes if we want to use special characters. Add **\n** for newlines and indent within the quotes to implement indentation. Text blocks solve this. They align based on the last line, so indentation is seamless. You can just use quotes wherever you want, and newlines are implicit. We use the formatted method of string to replace the two parameters, which makes it even more fluid.

There is even more on the horizon, JEP 430, and it is discussing an enhancement that goes beyond Strings. For example, refer to the following code:

```
1. ResultSet rs = DB."SELECT * FROM Person p WHERE p.last_name = \{name}";
```

We have not discussed databases yet, but the gist of this is that the DB class would be a processor that accepts parameterized Strings. This is not a string replacement like the example before but a completely new way of creating objects based on string parameters.

Records (Java 16)

We discussed records in the first two chapters of the book. They are remarkably powerful features that are relatively new to Java. With their introduction, we also gained static members for inner classes, which only allowed constants in the past.

Record patterns (Java 19 preview)

JEP 405 defines record patterns that correlate to the **instanceof** pattern-matching we discussed before. It enhances that with a syntax that extracts the values of the record. For example:

```
1. if (o instanceof Point(int x, int y)) {
2. System.out.println(x + y);
3. }
```

Notice that if **o** is an instance of **Point**, we would just have x and y from that Point object, and we could use them.

String templates (Java 21 preview)

JEP 430 defines string templates, which is designed to simplify string concatenation and enhance it. With the current version of Java, we have several approaches for building a string; perhaps the simplest would be as follows:

```
1. public String method(int arg) {
2. return "The value is: " + arg + " generated at: " + System.currentTimeMillis();
3. }
```

This is tedious, especially with text blocks (the “”” syntax). With String templates, we could write this as follows:

```
1. public String method(int arg) {
```

```
2. return STR."The value is: \{arg} generated at: \{System.currentTimeMillis()}";
3. }
```

This code is more fluid and readable and produces the same result. Notice the string is prefixed with STR. This is the template processor, and the beauty of this API is that it is pluggable. We can create our own template processors such as the following:

```
1. ResultSet rs = DB."SELECT * FROM Person p WHERE p.last_name = \{name}";
```

What we have here is a database processor that converts a parameter to a name. A simple read of the code might indicate there is an SQL injection bug in that code. That is not necessarily true. A string processor is under no requirement to replace the value in the string with the given value. It can create a proper **PreparedStatement** and verify that the value set in the name is indeed valid.

Unnamed patterns and variables (Java 21 preview)

A long-time limitation in Java is a lack of a simple way to remove redundancy. JEP 443 defines the underscore character as a placeholder for missing declarations in signatures. In current Java we would often write code like the following:

```
1. try {
2. // ....
3. } catch(Exception ex) {
4. genericErrorHandling();
5. }
```

Notice the **ex** variable is unused. This can produce a warning by code linting tools and can also be a bother as we pick names that pollute the code and appear when we search for things. This JEP provides a way to declare that we do not care about a particular value by using the underscore character as such:

```
1. try {
2. // ....
3. } catch(Exception _) {
4. genericErrorHandling();
5. }
```

The true value of this change becomes apparent when we combine it with

other language features, such as patterns and enhanced switch statements:

```
1. switch (b) {
2. case Box(RedBall _), Box(BlueBall _) -> processBox(b);
3. case Box(GreenBall _) -> stopProcessing();
4. case Box(_) -> pickAnotherBox();
5. }
```

We do not care about the parameter values, but we might care about the type. This saves us some typing in the first two switch statements but can work as a "catch all" in the last entry.

APIs

There were MANY API changes in the timeframe between Java 8 and 21. We could spend a chapter writing about the changes to the **String** class alone. These are some of the big-ticket changes that we need to understand. To see the full list of all changes, check out the Java Almanac[2].

HttpClient (Java 11)

There are many ways to make an HTTP request in Java, and they all have their faults. **HttpClient** is a major step forward in that regard. It is an immutable API that uses a very fluid syntax to perform a request. It has two versions of the API: synchronous and asynchronous. The synchronous API looks like the following:

```
1. HttpClient client = HttpClient.newBuilder()
2. .version(Version.HTTP_2)
3. .build();
4. HttpRequest request = HttpRequest.newBuilder()
5. .uri(URI.create("https://debugagent.com/"))
6. .build();
7. var response = client.send(request,
   HttpResponse.BodyHandlers.ofByteArray());
8. System.out.println(new String(response.body()));
```

As you can see, we just use builder patterns to define the request. The interesting bit is the response in Line 7. We can define any response handler out of a few "ready-made" options or a custom option of our own.

Foreign function and memory API—Panama (Java 19)

Java provides a lot of protection and comfort. But occasionally, we need to step outside of its warm embrace to invoke a native OS capability. In the past, Java standardized access to C using the **Java Native Interfaces** (**JNI**). It had some custom non-standard APIs too, but JNI was the leading standard. Unfortunately, JNI is widely despised.

It is hard to work with. It requires copying values back and forth. It requires writing a lot of C boilerplate. Naturally, third parties came along and offered various Java-based wrappers around JNI, but they are flawed since they are based on JNI. They implicitly inherited some of its biggest pain points.

Panama means overhauling access from the JVM to the native OS and potentially other languages. There are several JEPs involved since this is a huge undertaking, with some pieces already in preview and other pieces still pending. Panama can reduce some of the overhead of JNI and, as a result, can also provide better performance.

Because Panama deals with native code, we will not go deep into the API and will only provide a brief overview of its capabilities. We would normally use the **jextract** command line tool to convert a C header into Java stubs. At this point, we can make C calls directly from Java. For example, in the following hello world code, we invoke the native **printf** C function:

```
1. public class HelloPanama {
2. public static void main(String[] args) {
3. try (var confined = MemorySession.openConfined()) {
4. MemorySegment s = confined.allocateUtf8String("Hello Panama");
5. printf(s);
6. }
7. }
8. }
```

To allocate native memory, which we need to invoke a function, we need a memory session. Notice we need to clean up after ourselves, which is why we use try-with-resources. Check out the great series and talks by Carl Dea on this subject[3].

Another related part of Panama is support for the vector API, which will provide Java with primitives to support CPU vector primitives. CPUs are very good at processing data in batches; this makes vector operations (think matrix multiplications) very fast. However, we need to use specific CPU commands to get that $\times 16$ performance boost. Normally, the compiler does that for us, and sometimes it can. But there are many

nuances and no guarantees. The Vector API includes a set of primitives that force us to write code in a way that the JIT should recognize and translate to vector calls (SIMD, SSE, AVX, and so on). This API is in preview, but it is pending on Valhalla, as these classes should be primitive.

Structured concurrency (Java 19)

This API was introduced as part of JEP 428 and is closely tied to project Loom. It lets us limit two threads to a scope using an **AutoCloseable** interface and a try-with-resources block. When the following code is done, both threads will be completed:

```
try (ExecutorService e = Executors.newVirtualThreadPerTaskExecutor()) {
e.submit(() -> methodA());
e.submit(() -> methodB());
}
```

This effectively means that the close operation for the executor will act as a call to **join()**, which waits for the threads to complete.

Serialization filtering (Java 9)

One of the biggest security problems in many programming languages is serialization. It lets us read a bunch of bytes and convert them to an object. That might seem harmless, but unfortunately, that is not the case. A malicious hacker can use subtle mistakes in serialization code to craft a vulnerability chain. In such a chain, your small bug can allow the loading of another class, which points to another class down a long chain, until you accidentally load a vulnerable class.

There are developer tools that let hackers create such complex chains and even generate gadgets, which are effectively pre-packaged exploits. Unfortunately, these bugs are subtle and very common. *Brian Vermeer* called serialization “the gift that keeps giving” in his talks[4]. Leaving serialization unchecked is just asking for trouble.

Serialization filters are here to solve a significant part of the problem. They let us block specific classes or whitelist others. We can do this using the command line, or we can use code to block and monitor serialized classes. This is such an important API it was backported to older versions of the JDK!

The following command line filters out all serialization in the JVM:

```
java "-Djdk.serialFilter=!*" -jar myapp.jar
```

The exclamation point means we wish to block, and the star means we

block everything. We can explicitly block a specific package using the name of that package:

```
1. java "-Djdk.serialFilter=!mypackage.*" -jar myapp.jar
```

Or we can take the inverse route and explicitly allow that package and no other package:

```
1. java "-Djdk.serialFilter=mypackage.*;!*" -jar myapp.jar
```

This is all good, but what if we want to write code that filters or logs the serialization? We can do that globally for the entire JVM or for a specific stream using filter classes:

```
1. ObjectInputFilter.Config.setSerialFilter(info ->
2. info.depth() > 10 ? Status.REJECTED : Status.UNDECIDED);
```

This is a sample from the Oracle documentation of a simple serialization filter. Notice that it can reject the serialization or leave it undecided. This is part of a filter chain where each stage in the validation process can reject the serialization or pass it on to the next stage. We can bind the filter globally as we do here or do it on a per-stream basis. The API is remarkably flexible and provides a lot of information about the process.

Scoped values (Java 20)

Up until project Loom, we used thread locals to store thread-specific information. Say we logged into a secure system; we can use a thread local to save my login status. Then we do not have to pass around that status to every method. This is great and works with Loom as well, but it is not as convenient and is not as efficient as thread locals can be expensive to swap in and out.

JEP 429 introduced scoped values to give us the same benefits as thread locals, with lower overhead and cleaner syntax. The following code shows roughly how this works:

```
1. final static ScopedValue<ClassOfX> X = new ScopedValue<>();
2.
3. ScopedValue.where(X, valueOfX, () -> {
4. ClassOfX xValue = X.get();
5. // ...
6. });
```

This seems trivial. We can get X directly, so why do we need scoped value?

The difference is that the scoped value (like the thread local) is accessible down the chain in every method we invoke. That makes it in the scope of

the entire thread, not just the class or a method.

Sequenced collection (Java 21)

Java's collection framework includes a sophisticated and rich class hierarchy. However, it is missing a key interface within that hierarchy. **Collection** is the common base interface common within the class, but it is somewhat limited. It has no notion of order which classes like **ArrayList** and **Deque** include. **List** offers order, but it is too specific to the implementation of a list.

With JEP 431, we will have two additional interfaces in the hierarchy, one for the collection hierarchy:

```
1. interface SequencedCollection<E> extends Collection<E> {
2. // new method
3. SequencedCollection<E> reversed();
4. // methods promoted from Deque
5. void addFirst(E);
6. void addLast(E);
7. E getFirst();
8. E getLast();
9. E removeFirst();
10. E removeLast();
11. }
```

And another for the **Map** hierarchy:

```
1. interface SequencedMap<K,V> extends Map<K,V> {
2. // new methods
3. SequencedMap<K,V> reversed();
4. SequencedSet<K> sequencedKeySet();
5. SequencedCollection<V> sequencedValues();
6. SequencedSet<Entry<K,V>> sequencedEntrySet();
7. V putFirst(K, V);
8. V putLast(K, V);
9. // methods promoted from NavigableMap
10. Entry<K, V> firstEntry();
11. Entry<K, V> lastEntry();
12. Entry<K, V> pollFirstEntry();
13. Entry<K, V> pollLastEntry();
```

```
14. }
```

With these two interfaces, generic code can leverage interfaces higher up in the hierarchy without relying on the limited **Collection** interface or writing conditional code.

Future

In this section, we will discuss features that have not made it yet to the JDK.

GraalVM

GraalVM is well past its 20th version and is a part of OpenJDK. It is very much in the present. The reason it is in the future section is that it has yet to be the "main JVM".

When Sun Microsystems introduced the Java virtual machine, it was interpreted. It took a few years for Sun to evolve a proper JIT and create Hotspot, which is the current JIT. In fact, Hotspot has two JITs, one optimized for client base workloads (starts faster, less RAM) and one for server workloads (aggressive optimizations). Both are still powerful state-of-the-art JITs, but a lot has changed since the 90s.

GraalVM is a new type of VM built from the ground up. It started as a research project but has become a product and slowly gained traction. It can be used as a regular JVM and a JIT. But this point is often set aside in favor of its two unique properties:

- **Polyglot**: GraalVM is a polyglot VM. That means it can run other languages natively and connect between them. Java code and invoke Python, Ruby, JavaScript, and C. All can be hosted in a single VM.
- **Native Image**: GraalVM can package a Java application as a native image. That means it can compile a JAR to an OS native executable with no dependencies, a faster startup time, and low RAM overhead.

Both are amazing features, but the native image aspect is the one that most people care about. We will talk about packaging a native image with GraalVM when we discuss Spring Boot. In that chapter, we will also show the performance difference afforded by it.

Valhalla

Java has two types of variables: primitives and objects. Objects are always a reference, and primitives are always passed by value. It simplified a lot of things in Java. An **int** cannot be null. It is very fast, and it is almost free in terms of VM overhead. Unfortunately, some objects make more sense as

primitives. For example, the number wrapper objects are often there as placeholders for the primitives. Because we cannot have a List of primitives.

Furthermore, many objects do not make sense as an object. The dimension will never be null and will always have two integers. Why do we need the overhead of an object for that?

Valhalla wishes to cross that divide and let us write objects that act as primitives. This is one of the biggest changes to the Java language ever made, and it will require a major re-evaluation of many assumptions we have about code and the language. The first feature introduced by Valhalla is the **value** keyword which we can use like the following:

```
1. value class Dimension {
2. int width;
3. int height;
4. }
```

This means the class has no identity (pointer). It means the JVM will treat the class as a structure on the stack that can be copied but cannot be referenced. So, what will happen when we compare value classes if they have no identity?

```
1. if(dimension1 == dimension2) {
2. //...
3. }
```

This code will be the same as writing:

```
1. if(dimension1.width == dimension2.width && dimension1.height ==
   dimension2.height) {
2. //...
3. }
```

It will compare all the fields. This can have its advantages and its disadvantages. But this is a steppingstone. Value objects can still be null. Because they can still be null, they cannot be as efficient as primitives. That is why Valhalla includes the **primitive** keyword:

```
1. primitive class Dimension {
2. int width;
3. int height;
4. }
```

A primitive cannot be null. This means that assignment of values will work differently; the object will sometimes be copied in its entirety. This can

offer many advantages when done right. Refer to the following code:

```
1. var size = dimension[0];
```

This copies the primitive Dimension from the array. That means that changes to **size** will not impact **dimension[0]**. This is not very expensive since the copy would happen on the stack. It also means another thing in the **dimension[0]** array. The primitive array would be equivalent in terms of performance to an **int** array with twice the element count (because every dimension element has two integers). A standard Java array is made up of references. When traversing an array of objects and performing an operation, the VM needs to get the reference. Go through it to the object and perform the operation.

This can be very slow due to memory locality and paging, which can significantly impact time. With primitive objects, we can solve that, although value objects will help a lot along the way.

Conclusion

A lot has changed since Java 8. Even though Java 8 was a major release, the recent changes surpassed it overall and are worthy successors. Future capabilities such as Loom and Valhalla make the JVM exciting and more powerful than ever.

Fluidity and ease of use have been major themes in the language changes making their way into Java. Despite taking the “slow and steady wins the race” approach, Java is moving at a very fast pace.

Points to remember

- Use try-with-resources when working with closable resources.
- Switch expressions and pattern matching make it much easier to write fluid code.
- Text blocks can save a lot of pain when coding a string.
- When writing a data object, try to use records instead.
- If you are facing throughput issues, make sure to try your app with project Loom.
- HttpClient is a very powerful and simple API you can leverage for your networking.
- Always define a serialization filtering policy, always!
- GraalVM is a powerful tool you should keep an eye on. Benchmark it.
- Valhalla can have a significant impact on your project. Make sure you understand it and leverage it.

Multiple choice questions

A. Try with resources that helps you:

1. Catch a resource exception
2. Revert changes to a resource, for example, database rollback
3. Close a resource when you are done
4. Test various resources for compatibility

B. Project Loom is designed to improve:

1. Performance
2. Scale
3. Memory
4. Coding

C. GraalVM is a

1. JIT
2. Polyglot VM
3. Native Compiler
4. All of the above

D. A serialization filter can be assigned by:

1. Excluding patterns that are not supported
2. Including supported patterns
3. Via the command line
4. In code
5. All of the above

Answers

A. 3

B. 2

C. 4

1 https://github.com/shai-almog/java-book/

2 https://javaalmanac.io/jdk/21/apidiff/8/

3 https://foojay.io/today/project-panama-for-newbies-part-1/

4 https://www.youtube.com/watch?v=zHZv2L9hDis

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 4
Modern Threading

Introduction

At the time of writing this chapter, Java is poised to make another major change to its threading infrastructure with project Loom. This will be the third overhaul of the threading concept. However, there were many major features in Java that completely revolutionized the way we wrote concurrent code. Thread pools, **java.util.concurrent**, structured concurrency, and more.

In this chapter, we will discuss the evolution of Java threading from the beginning all the way to the future. We will explain the trade-offs and challenges, and we will also discuss the best ways to write highly concurrent code moving forward.

Structure

In this chapter, we will discuss the following topics:

- History
- Concepts
- Thread pools
- Locks
- Synchronizers
- Atomic
- Futures
- Collections and queues

Objectives

By the end of this chapter, you will understand the core concepts of

concurrency and parallelism—how they differ, how Java threading implementations changed, and what were the trade-offs. You will also understand the best ways to encapsulate your concurrent code.

History

Java 1.0 launched with green threads. To understand what that means, we need to understand how multi-threading works. There are two approaches for implementing multi-threading:

- **Cooperative**: Only one thread is running at a time. A thread needs to release control to a different thread.
- **Preemptive**: The operating system natively suspends a thread and cycles to the next thread.

Each approach has its advantages for different use cases. Green threads were implemented within the JVM and, as a result, were cooperative. Java 2 and other newer versions featured native threads. There were several reasons for that switch:

- **Resource usage**: Native threads can make use of OS and hardware resources. A native thread can run on a different CPU core; if a machine has multiple CPUs or cores, the additional threads can make use of them. Green threads are limited to a single core.
- **Native support**: Native APIs expect to work on a native OS thread and might need to control it. Using the Java thread might be a problem.
- **Resource Starvation:** CPU-intensive code could trigger resource starvation in the JVM without native threads. With native threads, the pre-emptive multi-threading can still function, even with a CPU-intensive thread.

This move was a smart one and enabled Java's high-performance status. However, native threads also have a problem. Every one of them has an overhead—roughly 2MB of overhead that cannot be controlled since the OS has no way of determining in advance how we wish to use the thread.

This worked reasonably well for many use cases. Most server applications could use a reasonable number of long-lived threads to perform tasks efficiently. However, there was one major use case that did not perform as nicely: a Web server.

On a Web server, an incoming request needs to process a task and perform another network request as a result (a database query is usually a network request). That means that the server spends time reading from a network socket and writing to/waiting for another network socket. During that

time, the thread is in a blocked state and cannot do anything else. This puts a hard limit on the scale of a Web server, as there is a limit on the number of threads. A Web server can be effectively idle and yet completely saturated simply because it ran out of native threads.

The workaround for this was asynchronous APIs that let us release the current thread while processing IO operations. This worked but made programming much harder since the code could no longer be linear. Every call had to go through a set of asynchronous callbacks to perform every task.

Project Loom introduced virtual threads to Java to solve this problem, as discussed in the previous chapter. Loom is an amazing innovation, but it is not ideal for all cases. You should not use virtual threads for things such as native method interaction, CPU-intensive code, and so on. *Figure 4.1* shows the road to the current state of threading in the JDK.

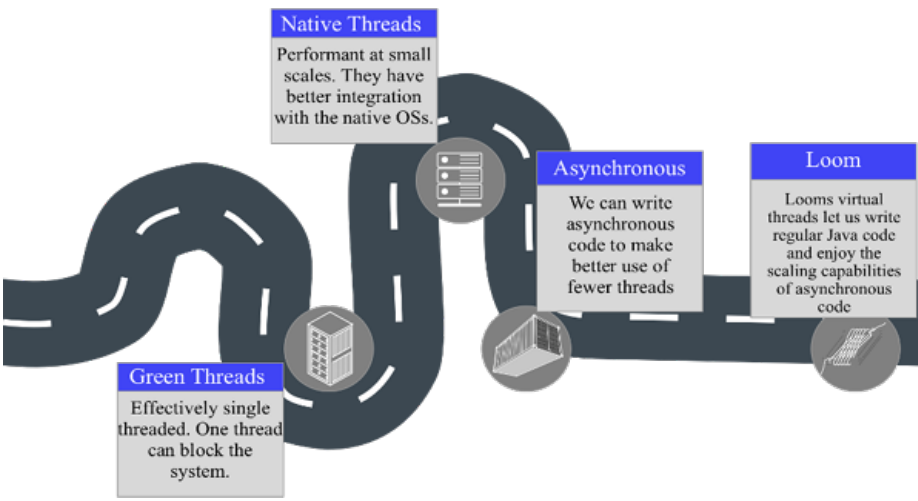
Native Threads
Performant at small scales. They have better integration with the native OSs.
Asynchronous
We can write asynchronous code to make better use of fewer threads
Loom
Looms virtual threads let us write regular Java code and enjoy the scaling capabilities of asynchronous code
Green Threads
Effectively single threaded. One thread can block the system.

Figure 4.1: *The road to modern threading in the JDK*

Concepts

Before we proceed, there are a few things we need to understand about parallelism, concurrency, and multi-threaded programming. Let us start with the difference between parallelism and concurrency. They are not interchangeable and describe two different situations.

Concurrency means that multiple threads are running. Green threads are concurrent as multiple threads are running concurrently. However, green threads never allowed two threads to execute at the same time. Even if your CPU has multiple cores, the green thread implementation allows for

only one thread.

Parallelism indicates that two or more threads execute (or might execute) at the same time. This requires a multi-core or multi-CPU system to leverage. Some failures can only occur when running in a parallel system, and as a result, it is much harder to write thread-safe code for a parallel system.

A parallel problem can be broken down into many smaller independent problems and run concurrently. This is expressed in the JDK by a parallel stream. Let us review the stream code we discussed in *Chapter 2, OOP Patterns*:

1. return numbers.stream()
2. .filter(num -> num % 2 == 0 && num > 0)
3. .distinct()
4. .sorted()
5. .collect(Collectors.toList());

We can change the **stream()** call to **parallelStream()**, which will perform the stream task in parallel using multiple threads and make better use of your CPU cores. In some situations, this will be noticeably faster than the regular stream, but it often will not since the overhead of multi-threading is higher than the reward for smaller workloads.

This works because streams have no external impact. We can take the different stages and run them independently of one another. If you wrote proper code that does not make use of external resources, this will work seamlessly.

Thread safety

The concept of thread safety is very complex. The basic definition of it is that an element can be accessed by multiple threads without worrying. This can be done, but this is not as trivial as it would seem. If a method changes a field within the class or reads a field from the class, thread safety might be compromised.

Historically, some operations in the JVM were not atomic. An atomic operation is guaranteed to be a single instruction. A thread will not be interrupted in the middle of an atomic operation. In those situations, if you set a long field to a new value, the thread might have been interrupted in the middle. To block that from happening, Java added the **volatile** keyword, which, when applied to a field, required all operations on it to be atomic.

This is (mostly) no longer a problem since we transitioned to 64-bit JVMs. In a 64-bit JVM, an operation on a 64-bit long value will be atomic. However, we said "mostly" because Valhalla adds new primitive types to the JVM, and they will not be atomic.

The original collections that shipped with JDK 1.0 tried to be thread-safe. However, they performed badly and had multiple thread-related issues. It is difficult to fully protect a complex mutable data structure. It is even harder to test and debug related failures. The new collection APIs abandoned thread safety for performance and offered specific patterns for thread-safe collections, which we will discuss soon. Today the common approach is to avoid thread-safety as much as possible. It slows down the application, even a multi-threaded application.

Mutex (Lock)

Mutex is shorthand for "mutual exclusion". It forces a limit on a shared resource whereby only a specific thread is allowed access. Java's "classic" mutex is the synchronized keyword. Synchronized can be used in multiple ways; a common way is as follows:

```
1. private Collection<String> listOfNames = new ArrayList<>();
2.
3. public synchronized void addName(String name) {
4. listOfNames.add(name);
5. }
6.
7. public synchronized boolean isInList(String name) {
8. return listOfNames.contains(name);
9. }
```

Every **Object** in Java carries a mutex. In this case, the **synchronized** keyword applies to the current object: **this**. Had the method been static, it would have applied to the corresponding **Class** object. If two threads reached these methods at once, only one would "get in". The other would have to wait until completion. Note that this applies to both methods. If a thread is currently within **addName**, calls to both **isInList** and **addName** would be blocked until it returns.

This type of programming has elegance, as we declare our thread safety behavior in the method signature. But most developers shun this way of programming as it is very problematic. A third-party developer can synchronize on the object and create unpredictable behavior, which is why synchronizing on the current object is heavily discouraged. We can solve this by using synchronized blocks as follows:

```
1. private final Object LOCK = new Object();
2. private Collection<String> listOfNames = new ArrayList<>();
3.
4. public void addName(String name) {
5. synchronized (LOCK) {
6. listOfNames.add(name);
7. }
8. }
9.
10. public boolean isInList(String name) {
11. synchronized (LOCK) {
12. return listOfNames.contains(name);
13. }
14. }
```

This is a more conventional approach to synchronization that does not expose the lock object. For trivial use cases, this can work, but for complex options, we will run into limitations. Say we want to add an event whenever someone adds a name to the list. That way, we can update the UI to indicate the latest person to join. Other parts of the code might also require that event, so we will write something generic like the following:

```
1. private Collection<String> listOfNames = new ArrayList<>();
2. private Collection<NameAdded> listeners = new ArrayList<>();
3.
4. interface NameAdded {
5. void nameAdded(String name);
6. }
7.
8. public void listenNameIsAdded(NameAdded callback) {
9. synchronized (LOCK) {
10. listeners.add(callback);
11. }
12. }
13.
14. public void stopListeningNameIsAdded(NameAdded callback) {
15. synchronized (LOCK) {
16. listeners.remove(callback);
17. }
```

```
18. }
19.
20. public void addName(String name) {
21. synchronized (LOCK) {
22. listOfNames.add(name);
23. for(NameAdded current : listeners) {
24. current.nameAdded(name);
25. }
26. }
27. }
```

Every time a name is added, we loop over the listeners and notify them all about the event. There are several problems with this code, but the biggest one is its potential for a deadlock. The big problem is that we are taking the thread that invoked **addName** and using it to "send an event". We invoke multiple callback methods that might do anything. Because it is holding the lock while doing that, no other method can run during that time. This means that the performance will be very slow and unpredictable.

Wait and notify (monitor)

Typically, events are sent on a different thread. We cannot send a thread per-event, so the solution is to send the events on a single thread. But that is a bit of a problem; how do we send data to another thread efficiently?

How do we keep and reuse the same thread over and over?

There are many ways to do that. We will start with the "old way", which will help us understand the new way better when we get to it:

```
1. private final Object EVENT_THREAD_LOCK = new Object();
2. private List<String> pending = new ArrayList<>();
3. public WaitAndNotify() {
4. new Thread(() -> {
5. try {
6. while(true) {
7. String[] newStrings;
8. NameAdded[] listenersArray;
9. synchronized (EVENT_THREAD_LOCK) {
10. if(pending.isEmpty()) {
11. EVENT_THREAD_LOCK.wait();
```

```
12. }
13. newStrings = new String[pending.size()];
14. pending.toArray(newStrings);
15. pending.clear();
16. }
17. synchronized (LOCK) {
18. listenersArray = new NameAdded[listeners.size()];
19. listeners.toArray(listenersArray);
20. }
21.
22. for(String name : pending) {
23. for(NameAdded nameAdded : listenersArray) {
24. nameAdded.nameAdded(name);
25. }
26. }
27. }
28. } catch (InterruptedException e) {
29. e.printStackTrace();
30. }
31. }).start();
32. }
33.
34. public void addName(String name) {
35. synchronized (LOCK) {
36. listOfNames.add(name);
37. }
38. synchronized (EVENT_THREAD_LOCK) {
39. pending.add(name);
40. EVENT_THREAD_LOCK.notify();
41. }
42. }
```

There is a lot to digest in this listing. We have a new constructor that sets up a new thread in the beginning. That part should be self-explanatory. We have two new fields; one is the pending names for which an event was not yet sent. The other is an additional lock for the new thread.

We do not want to reuse the same lock since that will increase the chance of a deadlock and will reduce performance.

The new thread is an infinite loop. The important part is in Line 11. We invoke the method **wait()**. With **wait()**, we can hold the thread indefinitely (as is the case here) or wait for a few milliseconds. To "wake up" the thread, we invoke the method **notify()**, as we can see in line 40. Both **wait()** and **notify()** require a lock on the object. If they were not within a synchronized block of the same object on which they were invoked, they would not work. When a thread is "waiting", it releases the surrounding lock, and that is why the **addName()** method can synchronize on the event thread lock.

This lets us park a thread and then wake it up when we need to. We can "park" multiple threads and wake them all up with **notifyAll()**; in this case, there is no need for that. The rest of the code is tedious but crucial. To keep code efficient, we want to minimize the scope of the synchronized block. Unfortunately, this means we cannot access variables as they might change. The common trick is to copy them to a separate object, which is what we do here. We can then fire the event from the thread and go back to sleep if there are no further events to process.

Deadlock

Two of the most common issues in thread programming are deadlocks and race conditions. We will start with the former. A deadlock occurs when a thread tries to gain access to a lock that another thread will not release. Typically, this would happen when one thread is waiting for the other and vice versa, as is the case in the following example, which is one of the simplest versions of a deadlock:

```
1. private static void dead(String name, Object lock1, Object lock2) {
2. try {
3. synchronized (lock1) {
4. System.out.println("Thread " + name + " Holding...");
5. Thread.sleep(2);
6. synchronized (lock2) {
7. System.out.println("Thread " + name + " Got it!");
8. }
9. }
10. } catch (InterruptedException e) {
11. throw new RuntimeException(e);
12. }
```

```
13. }
14.
15. public static void main(String[] argv) {
16. Object LOCK1 = new Object();
17. Object LOCK2 = new Object();
18. Executor executor = Executors.newFixedThreadPool(2);
19. executor.execute(() -> dead("One", LOCK1, LOCK2));
20. executor.execute(() -> dead("Two", LOCK2, LOCK1));
21. }
```

In Lines 19 and 20, we create two threads with one small difference. The first thread grabs lock1 first and lock2 second. The second thread grabs lock2 first and then lock1.

Both threads end up stuck in Line 6, each one waiting for the other thread to release the lock. That might sound contrived, but it is a very common problem when we start having locks all over the place. The “solution” of one lock would make deadlocks less likely, but would also slow down the application considerably. Furthermore, if you do have a deadlock, it would be a terrible one since the entire application will be blocked.

Race conditions

A race condition is a devious bug. It does not always happen. In fact, the following code worked despite having an obvious race condition in it:

```
1. private static List<String> stringList;
2. public static void main(String[] argv) {
3. Executors.newFixedThreadPool(1).
4. execute(() -> {
5. System.out.print(stringList);
6. });
7. stringList = List.of("ZZ", "XX", "LL", "DDD");
8. }
```

You will notice that **stringList** is initialized only after the executor is launched. This works because the thread takes a while to initialize, and by the time it is done, the list is ready. Despite that, we cannot rely on such behaviors. Conditions change, and a bug that does not happen at this moment might happen later.

Thread pools

We worked already with thread pools to some degree. They are a part of the **Executor** API that we saw in the demos. One of the costly aspects of threading is spinning up a new thread. Therefore, Java 5 added a standardized way to keep a fixed number of threads for reuse. This was a pattern that was already common in the Java community beforehand. A thread pool is designed to keep us from using too many threads. It is designed to limit our abuse of the system and force us to use it responsibly.

Notice that thread pools only make sense for native threads. The overhead of a Loom virtual thread is so low that a thread pool would reduce performance.

In the following code, it seems that we create 20,000 threads. We actually do not. Twenty threads are created, and the rest end up queued. Then, as each thread finishes, the next task in line is executed by the thread that became available:

```
1. var pool = Executors.newFixedThreadPool(20);
2. for(int iter = 0 ; iter < 20000 ; iter++) {
3. var currentThread = iter;
4. pool.execute(() -> System.out.println(
5. "In thread " + currentThread));
6. }
```

The result is similar to running with 20,000 threads, except for the fact that it is faster.

Executors

The **Executors** class is a helper class with static methods that help us create the type of pools we need. We used it before when creating virtual threads, and we can use it for a fixed-size queue. Most methods in the class return an **Executor** sub-interface; it is the abstraction representing a thread. We will discuss it more in-depth soon.

The **Executors** class has several factory methods for creating **Executor** objects, including:

- **newFixedThreadPool(int nThreads)**: Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. We saw this method in action in the previous sample. At any point, at most **nThreads** threads will be actively processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.
- **newSingleThreadExecutor()**: Creates an executor that uses a single worker thread operating off an unbounded queue. Tasks are

guaranteed to execute sequentially, and no more than one task will be active at any given time.

- **newCachedThreadPool()**: Creates a thread pool that creates new threads as needed but will also reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks.
- **newScheduledThreadPool()**: Creates a **ScheduledThreadPoolExecutor** that uses a fixed number of threads to execute tasks but also allows you to schedule tasks for future execution using a delay or a fixed rate. For example, this code will schedule the runnable for execution in 10 seconds: **executor.schedule(runnable, 10, TimeUnit.SECONDS);**
- **newVirtualThreadPerTaskExecutor()**: Creates a virtual thread; we discussed this when covering project Loom in the previous chapter.

There are some fantastic capabilities in the executor APIs. We will discuss some of the more interesting features in the following section.

Locks

Synchronized was an innovative and powerful feature of the Java language. But it has many drawbacks, some of them related to its age and others related to the fact that as a language feature, it is harder to evolve it as opposed to evolving an API. The concurrency APIs in Java include several amazing tools meant to reduce the need for synchronization. They are less error-prone, more elegant, and in some cases, even more efficient. The advantages of the lock are expressed in the following table:

	Synchronized	Lock
Simple to block in one method and released in another	Impossible	Possible
Optional	None	Try lock
Try lock	No	Yes
Interruptible waiting	No	Yes

Table 4.1: *Advantages of the lock*

Some of the elements in the preceding *table 4.1* are things we did not explain yet. Let us start by adapting some synchronized code to use locks. This is based on our second synchronization sample:

```
1. private final ReentrantLock LOCK = new ReentrantLock();
2. private Collection<String> listOfNames = new ArrayList<>();
3.
4. public void addName(String name) {
```

```
5. LOCK.lock();
6. try {
7. listOfNames.add(name);
8. } finally {
9. LOCK.unlock();
10. }
11. }
12.
13. public boolean isInList(String name) {
14. LOCK.lock();
15. try {
16. return listOfNames.contains(name);
17. } finally {
18. LOCK.unlock();
19. }
20. }
```

Lock is an interface that implements this API. We used **ReentrantLock**, which has semantics that most closely matches the **synchronized** keyword. Notice that the lock version is slightly longer due to the use of try and catch. We could have used a terse syntax such as the following:

```
1. public void addName(String name) {
2. LOCK.lock();
3. listOfNames.add(name);
4. LOCK.unlock();
5. }
```

However, this is risky. If the **add** method throws an exception, the lock would remain locked, and the method (or any other block that uses that lock) would become unusable. This is something the synchronized block handled for us.

You might be looking at that code and thinking, why can we not use try-with-resources like the following:

```
1. public void addName(String name) {
2. try(LOCK.lock()) {
3. listOfNames.add(name);
4. }
5. }
```

This would be nice and on par with a synchronized block. **Unfortunately, locks are not *AutoClosable* and cannot be unlocked seamlessly**. The API was designed for resources that "close" and a lock remains. We will reuse the lock object, but we will not reuse the file stream. This is doable by wrapping the lock object.

There are several interesting features for the lock we discussed previously. The first one is fairness. A **ReentrantLock** can be made fair by constructing it with true as an argument: **new ReentrantLock(true)**. Fairness means the thread that is waiting for the longest for the lock will be the next one to enter the lock. Synchronized makes no such guarantee. Notice that this does not guarantee that things will be in order. Take into consideration the following code:

```
1. private static final ReentrantLock FAIR_LOCK = new ReentrantLock(true);
2. private static final ReentrantLock UNFAIR_LOCK = new ReentrantLock();
3.
4. private static void print(Lock lock, int number) {
5. lock.lock();
6. try {
7. System.out.print(" " + number);
8. } finally {
9. lock.unlock();
10. }
11. }
12.
13. private static void print(Executor executor, Lock lock) {
14. for(int iter = 0 ; iter < 10 ; iter++) {
15. var current = iter;
16. executor.execute(() -> print(lock, current));
17. }
18. }
19.
20. public static void main(String[] argv) throws InterruptedException {
21. Executor executor = Executors.newFixedThreadPool(10);
22. System.out.println("Fair: ");
23. print(executor, FAIR_LOCK);
24. Thread.sleep(10);
25. System.out.println("\nUnfair: ");
```

```
26. print(executor, UNFAIR_LOCK);
27. Thread.sleep(10);
28. }
```

One might assume that fair locking will always produce numbers 0 to 9 in the natural order. While unfair locking would not. However, this is not necessarily the case. The threads might reach the lock in a different order than their creation order, and thus, the fairness will not match the creation order. Fairness is an important property but not something one can rely on for most purposes.

Interrupting threads is not a new feature. However, locks introduced a new ability to interrupt the thread in the queue:

```
1. private static final ReentrantLock LOCK = new ReentrantLock();
2. private static Thread waitingThread;
3.
4. private static void interruptedLock(int number) {
5. try {
6. LOCK.lockInterruptibly();
7. } catch (InterruptedException err) {
8. System.out.println("Interrupted thread: " + number);
9. return;
10. }
11. try {
12. System.out.println("Entered thread " + number);
13. Thread.sleep(30);
14. System.out.println("Exiting thread "+ number);
15. } catch (InterruptedException err) {
16. System.out.println("Sleep Interrupted: " + number);
17. } finally {
18. LOCK.unlock();
19. }
20. }
21.
22. public static void main(String[] argv) throws InterruptedException {
23. Executor executor = Executors.newFixedThreadPool(10);
24. executor.execute(() -> interruptedLock(1));
25. executor.execute(() -> {
```

```
26. waitingThread = Thread.currentThread();
27. interruptedLock(2);
28. });
29. while(!LOCK.hasQueuedThreads()) {
30. Thread.yield();
31. }
32. waitingThread.interrupt();
33. }
```

In line 24, we start the first thread that grabs the lock. Notice that **Thread.sleep()** does not release the lock. Under normal circumstances, you do not want to do that code; we are using it here to simulate a long-running task. Line 25 launches another thread that will now block on that lock. In line 29, we wait for the thread to enter the lock queue and then send an interrupt. Notice that this will not work for a regular **lock()** invocation; it only works for **lockInterruptibly()** as used in Line 6. The resulting output is as follows:

Entered thread 1

Interrupted thread: 2

Exiting thread 1

Notice that the second thread was interrupted during the locking phase. This lets us detect a case where a thread is blocked on a lock and "rescue" that thread. We can go further by using **tryLock()** as such:

```
1. private static final ReentrantLock LOCK = new ReentrantLock();
2.
3. private static void tryLock(int number) {
4. try {
5. LOCK.tryLock(10, TimeUnit.MILLISECONDS);
6. } catch (InterruptedException err) {
7. System.out.println("Interrupted thread: " + number);
8. return;
9. }
10. try {
11. System.out.println("Entered thread " + number);
12. Thread.sleep(30);
13. System.out.println("Exiting thread "+ number);
14. } catch (InterruptedException err) {
```

```
15. System.out.println("Sleep Interrupted: " + number);
16. } finally {
17. LOCK.unlock();
18. }
19. }
20.
21. public static void main(String[] argv) throws InterruptedException {
22. Executor executor = Executors.newFixedThreadPool(10);
23. executor.execute(() -> tryLock(1));
24. executor.execute(() -> tryLock(2));
25. }
```

This code has the same output as the previous listing. The first thread grabs the lock and waits. The second thread tries to lock and blocks for 10 milliseconds. It then times out and returns. **tryLock()** can be used without an argument; it would return false if it failed to grab the lock. This lets us handle contention intelligently. A good example of such a case would be a situation where a process needs to change three separate elements; we can try to lock each one and make better use of our time.

Up until now, we focused on **ReentrantLock**, which offers some improvements to synchronized while keeping a similar set of semantics. We also have **ReadWriteLock**, which is implemented by **ReentrantReadWriteLock**. This type of lock is in fact two separate locks. One for reading and one for writing. For example, refer to the following code snippet:

```
1. private final ReadWriteLock LOCK = new ReentrantReadWriteLock();
2. private Collection<String> listOfNames = new ArrayList<>();
3.
4. public void addName(String name) {
5. LOCK.writeLock().lock();
6. try {
7. listOfNames.add(name);
8. } finally {
9. LOCK.writeLock().unlock();
10. }
11. }
12.
13. public boolean isInList(String name) {
```

```
14. LOCK.readLock().lock();
15. try {
16. return listOfNames.contains(name);
17. } finally {
18. LOCK.readLock().unlock();
19. }
20. }
```

In the previous synchronization code, we had one lock. That is not as efficient as it could be. When adding a name, we always need to lock it. However, when we can have multiple threads reading from the list at once, there will be no overhead for that. The reason we do that locking is that during list modification, we cannot read from it. The **ReadWriteLock** solves that common situation. Multiple threads can hold the read lock and run concurrently with no issue. If you try to obtain a write lock, the read threads need to be released.

While the write lock is held, all read operations will also be blocked. This is far more efficient as read operations are often more common than write operations. As a result, performance can be much faster than the original naïve code.

Still, the **ReentrantReadWriteLock** suffers from many shortcomings. If you obtain a read lock, you cannot "upgrade" it. Let us say we want to keep names unique in the preceding list. We do not necessarily need a write lock. The name might already be in the list, in which case, we do not need a write lock. However, if we do end up needing a write lock, we will not be able to get it without relinquishing the read lock first. It does not support optimistic reads and is not as efficient as it can be.

Java 8 solved all these limitations by introducing the **StampedLock**. The API for this type of lock is slightly different than the other locks and provides the ability to upgrade or downgrade the lock, as shown follows:

```
1. private final StampedLock LOCK = new StampedLock();
2. private Collection<String> listOfNames = new ArrayList<>();
3.
4. public void addName(String name) {
5. long stamp = LOCK.readLock();
6. try {
7. if(!listOfNames.contains(name)) {
8. long writeLock = LOCK.tryConvertToWriteLock(stamp);
9. if(writeLock == 0) {
```

```
10. throw new IllegalStateException();
11. }
12. listOfNames.add(name);
13. }
14. } finally {
15. LOCK.unlock(stamp);
16. }
17. }
18.
19. public boolean isInList(String name) {
20. long stamp = LOCK.readLock();
21. try {
22. return listOfNames.contains(name);
23. } finally {
24. LOCK.unlockRead(stamp);
25. }
26. }
```

Unlike standard locks, the **StampedLock** returns a long value that we need to use to release the lock. If you are using Java 8 or newer, you should probably use this API.

There is one last feature of synchronized that we did not cover: **wait/notify**. Locks support this through the Condition API. Notice that conditions are not supported by **StampedLock**, which is where **ReentrantLock** variants are still superior.

The event system we outlined before can be adapted to use conditions with code such as the following:

```
1. private final ReadWriteLock LOCK = new ReentrantReadWriteLock();
2. private final Condition NOTIFICATION = LOCK.readLock().newCondition();
3. private Collection<String> listOfNames = new ArrayList<>();
4. private Collection<NameAdded> listeners = new ArrayList<>();
5.
6. private List<String> pending = new ArrayList<>();
7.
8. public Conditions() {
9. new Thread(() -> {
10. try {
```

```
11. while(true) {
12. String[] newStrings;
13. NameAdded[] listenersArray;
14. LOCK.writeLock().lock();
15. try {
16. if(pending.isEmpty()) {
17. NOTIFICATION.await();
18. }
19. newStrings = new String[pending.size()];
20. pending.toArray(newStrings);
21. pending.clear();
22. } finally {
23. LOCK.writeLock().unlock();
24. }
25. synchronized (LOCK) {
26. listenersArray = new NameAdded[listeners.size()];
27. listeners.toArray(listenersArray);
28. }
29.
30. for(String name : pending) {
31. for(NameAdded nameAdded : listenersArray) {
32. nameAdded.nameAdded(name);
33. }
34. }
35. }
36. } catch (InterruptedException e) {
37. e.printStackTrace();
38. }
39. }).start();
40. }
41.
42. interface NameAdded {
43. void nameAdded(String name);
44. }
45.
46. public void listenNameIsAdded(NameAdded callback) {
```

```
47. LOCK.writeLock().lock();
48. try {
49. listeners.add(callback);
50. } finally {
51. LOCK.writeLock().unlock();
52. }
53. }
54.
55. public void stopListeningNameIsAdded(NameAdded callback) {
56. LOCK.writeLock().lock();
57. try {
58. listeners.remove(callback);
59. } finally {
60. LOCK.writeLock().unlock();
61. }
62. }
63.
64. public void addName(String name) {
65. LOCK.writeLock().lock();
66. try {
67. listOfNames.add(name);
68. pending.add(name);
69. NOTIFICATION.signal();
70. } finally {
71. LOCK.writeLock().unlock();
72. }
73. }
74.
75. public boolean isInList(String name) {
76. LOCK.readLock().lock();
77. try {
78. return listOfNames.contains(name);
79. } finally {
80. LOCK.readLock().unlock();
81. }
82. }
```

The code is simplified by unifying the locks. The thing to notice is that since a lock is a Java object, it will have the **wait()** and **notify()** methods. This is very confusing since we should avoid them. Instead, we need to use the **signal()** and **await()** methods. There are some additional minor differences, but the overall feature is not that interesting.

This is a very complicated approach to implement the equivalent of **executor.execute(Runnable)**. We should use **execute**; the value of this code is in understanding the heavy lifting performed by that API.

Synchronizers

Several classes in the concurrency package fall under the banner of synchronizers that are not locked. The most classic among them is the **Semaphore**, which is a classic concept in concurrent programming. A Semaphore is a system of flags used to signal information in the old days. Most importantly, a semaphore is raised to stop a train from entering a busy station.

The **Semaphore** class in Java limits the number of threads that can enter a block. This is useful if a resource can become a bottleneck. A good example would be a database that has a limit on the number of concurrent users. We can use a Semaphore to restrict access and prevent overuse:

```
1. private final Semaphore limited = new Semaphore(20);
2. private void restricted() throws InterruptedException {
3. limited.acquire();
4. try {
5. // perform restricted activity
6. } finally {
7. limited.release();
8. }
9. }
```

The **CountDownLatch** can wait until the count reaches zero. Imagine a situation where we have multiple threads that need to complete tasks so that our code can proceed. How do we know that they are all finished?

We use the **CountDownLatch** to count, and when it reaches zero, the **await()** method will release our current thread and indicate that we are all done:

```
1. Executor executor = Executors.newFixedThreadPool(10);
2. CountDownLatch latch = new CountDownLatch(10);
3. for(int iter = 0 ; iter < 10 ; iter++) {
```

```
executor.execute(() -> {
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
throw new RuntimeException(e);
}
latch.countDown();
});
}
latch.await();
System.out.println("All 10 threads completed");
```

The **Exchanger** class enables two threads to exchange objects. It provides a synchronization point where two threads can pair and swap elements within those threads. Here is an example of two threads passing data in both directions, for example, a thread that sends commands to another thread and processes results:

```
Exchanger<String> exchanger = new Exchanger<>();
Executor executor = Executors.newFixedThreadPool(2);

// send commands
executor.execute(() -> {
try {
String result = exchanger.exchange("List Variables");
} catch (InterruptedException e) {
throw new RuntimeException(e);
}
});

// send perform command and return results
executor.execute(() -> {
try {
String command = exchanger.exchange(null);
} catch (InterruptedException e) {
throw new RuntimeException(e);
}
});
```

Atomic

The atomic package contains multiple atomic classes for handling a single variable “atomically”, without risking concurrency issues. This is typically the case for primitives in Java, especially if they are marked as volatile. However, this is not guaranteed when multiple threads change values concurrently.

The following code is the best possible way to demonstrate how important it is to use an atomic variable:

```
1. private static long regular;
2. private static volatile long vol;
3. private static Long obj = 0L;
4. private static volatile Long volObj = 0L;
5. private static AtomicLong atomicLong = new AtomicLong(0);
6. public static void main(String[] argv) throws InterruptedException {
7. Executor executor = Executors.newFixedThreadPool(20);
8. CountDownLatch countDownLatch = new CountDownLatch(20);
9. for(int iter = 0 ; iter < 20 ; iter++) {
10. executor.execute(() -> {
11. for(int x = 0 ; x < 1_000_000 ; x++) {
12. regular++;
13. vol++;
14. obj = obj++;
15. volObj = volObj++;
16. atomicLong.incrementAndGet();
17. }
18. countDownLatch.countDown();
19. });
20. }
21. countDownLatch.await();
22. System.out.println("regular: " + regular + " volatile: " + vol + "
    object: " + obj +
23. " volatile object: " + volObj + " atomic: " + atomicLong.get());
24. }
```

On a Mac Book Air M2 machine, the output was the following:

regular: 10917365 volatile: 10957373 object: 51921 volatile object:

422968 atomic: 20000000

Notice the enormous difference between the correct atomic result at the end and all the results that preceded it. This is especially true for object types that performed particularly badly.

Synchronization or locks can solve these problems, but this package is simpler to use.

Futures

The **notify()** and **Condition** code from before is bad. It is long, verbose, and hard to understand. It would be even worse if we needed to return a value. There has got to be a better way to spin something into a separate thread and get a result. The good news is that there is one such answer to my problems; it is a system that is common across many programming languages, known by the name Future and Promise. The Java promise API is named **CompletableFuture**, and it was introduced as part of Java 8.

A Future represents the result of an asynchronous computation. A **CompletableFuture** (also known as a promise) is a future that can be explicitly completed. In the following sample, we run on a separate thread and return the result:

```
1. Executor executor = Executors.newSingleThreadExecutor();
2. Supplier<String> r = () -> {
3. try {
4. return Files.readString(new File("pom.xml").toPath());
5. } catch (IOException e) {
6. throw new CompletionException(e);
7. }
8. };
9. var future = CompletableFuture.supplyAsync(r, executor);
10. System.out.println(future.get());
```

We create a single-thread executor, which means that all operations will run on one thread. We could use a thread pool or a virtual thread executor to create multiple threads. Notice that the executor argument in Line 9 is optional.

However, this is only a teaser of the full power of this API. The challenge of asynchronous code is in dependencies. We often want to implement a flow that starts in one thread, moves to another thread, and asynchronously returns the result to a final location. There is one limitation, though; you will notice that the previous code had a try/catch

block that was not really handled. Where did that exception "go"?

This is where stages come in, and a **CompletableFuture** is comprising of multiple **CompletionStage** instances that we can assemble using a builder pattern. We can daisy chain them to create a process that implements multiple stages. A simple example of this could be as follows:

```
1. CompletableFuture.supplyAsync(r, executor)
2.         .exceptionally(err -> "Error reading file: " + err)
3.         .thenAcceptAsync(value -> System.out.println(value));
```

However, there can be far more elaborate examples, such as **CompletableFuture.allOf(...)**, which binds together multiple completable futures into a single operation that we can treat as a single future. This does not return a result but helps us wait for multiple futures to complete by using the **join()** method.

Collections and queues

One of the most common problems in concurrency is for shared sets of elements. Java 1.0 had some collection classes, specifically **Vector** and **Hashtable**. Both tried to be thread-safe, and both did not do a great job of it. Their use is discouraged in modern Java. There are solutions for making the existing collection classes "seem" thread-safe, specifically **Collections** wrapper methods: **synchronizedCollection**, **synchronizedList**, and **synchronizedMap**. Those are all decent solutions for simple use cases. However, if a collection needs to scale to high throughput and more nuanced support, then this might not be enough.

The **CopyOnWriteArrayList** and **CopyOnWriteArraySet** work as a List or Set implementation. They are mainly useful for a collection in which very few mutations are made but read access must be *very* fast. In these collections, the access to the elements is fast and unsynchronized. Mutating operations change a different data structure which then replaces the entire list or set in one atomic operation. This means changes are significantly slower than a synchronized **ArrayList** or **ArraySet**. However, reads are almost as fast as an unsynchronized version. It is hard to tell if this is a trade-off that makes sense for most developers, so you will need to benchmark your code to make a call.

These and the other collections in the concurrent package do not throw a **ConcurrentModificationException** and might allow a stale iterator to proceed.

ConcurrentLinkedQueue is a thread-safe, unbounded queue that uses linked nodes to store its elements. It is an implementation of the **BlockingQueue** interface. One of the main advantages of

ConcurrentLinkedQueue is that it does not require any locking or synchronization to add or remove elements from the queue. This makes it a very fast and efficient queue implementation, especially in multi-threaded environments, where many threads may be accessing the queue concurrently.

There are several additional concurrent collection classes in Java that are designed for use in multi-threaded environments. Here is a brief overview of some of the most used concurrent collections:

- **ConcurrentHashMap**: This is a thread-safe implementation of the Map interface that uses a hash table data structure to store its elements. It is similar to the **HashMap** class, but it uses locks to ensure that only one thread can modify the map at a time.
- **ConcurrentLinkedQueue**: As we discussed previously, this is a thread-safe, unbounded queue that uses linked nodes to store its elements. It is an implementation of the **BlockingQueue** interface and is based on a linked list data structure. It is a very fast and efficient queue implementation, especially in multi-threaded environments.
- **ConcurrentLinkedDeque**: This is a thread-safe, unbounded double-ended queue that uses linked nodes to store its elements. It is an implementation of the Deque interface and is based on a linked list data structure. It supports efficient insertion and removal of elements at both ends of the queue.
- **ConcurrentSkipListMap**: This is a thread-safe implementation of the **SortedMap** interface that uses a skip list data structure to store its elements. It is similar to the **TreeMap** class, but it uses locks to ensure that only one thread can modify the map at a time.
- **ConcurrentSkipListSet**: This is a thread-safe implementation of the **SortedSet** interface that uses a skip list data structure to store its elements. It is similar to the **TreeSet** class, but uses locks to ensure that only one thread can modify the set at a time.

It is worth noting that these concurrent collection classes are not suitable for all use cases. For example, if you need to support very high levels of concurrency (for example, thousands of threads), it may be more efficient to use a lock-free data structure such as a **ConcurrentHashMap** or **ConcurrentLinkedQueue**. On the other hand, if you only need to support a small number of concurrent threads, a synchronized collection wrapper using the **Collections** class may be sufficient. There are many other queue and collection classes in the concurrency package. Each has its nuanced trade-offs that we need to be aware of.

Conclusion

The threading support in Java has evolved dramatically over the past decades. Java leaped to the forefront of the industry by virtue of its built-in threading support. However, it fell behind as that support became a hindrance. Modern versions of the API fixed that and currently offer unrivaled constructs for dealing with concurrency.

The core implementation of threads was an amazing innovation, but it too introduced a problem of scale. The introduction of project Loom is changing that and, once again, brings Java to the technological forefront.

Points to remember

- Use locks instead of the synchronized keyword.
- Benchmark and test concurrent collection to verify performance in the appropriate scale.
- Chain concurrent operation dependencies using **CompletableFuture**.
- Use atomic variables for consistency when applicable.
- Whatever you need to do, there is probably a built-in class that already does that.

Multiple choice questions

A. Which is the best in terms of performance for a read/write lock?

1. Synchronized
2. Synchronized block
3. ReentrantLock
4. ReentrantReadWriteLock
5. StampedLock

B. When should we pick a native thread over a virtual thread?

1. High CPU usage
2. Non-network operation
3. Call native code
4. All the above

C. CopyOnWriteArrayList is better than a synchronized List…

1. Always

2. High read volume low write
3. High write volume low read
4. Small lists
5. Large lists

D. Which of these can replace an atomic wrapper?

1. Lock
2. Semaphore
3. CountDownLatch
4. volatile
5. volatile but only on a primitive

Answers

A. 5

B. 4

C. 2

D. 1

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 5
It's Springtime in Java

Introduction

Spring is the leading Java backend framework. It became the *de facto* standard for Java backend development by making some bold architectural choices and offering sophisticated deployment options. There are many backend frameworks for Java, but Springs usage exceeds all of them combined![1]

Spring brings together libraries and tools from multiple different projects to create a unified experience. Spring is also highly configurable; it is not limited to the backend or database-driven applications. However, to cover the depth and breadth of Spring, we would need a much bigger, dedicated book.

Structure

In this chapter, we will discuss the following topics:

- History and origin
- Inversion of Control, Dependency Injection, and Aspect Oriented Programming
- Hello Spring Boot
- A REST[2] API
- Spring MVC and Thymeleaf
- SQL and JDBC
- JPA
- Error handling

Objectives

By the end of this chapter, you will be able to build a simple database-driven application with a Web API. You will understand the basic concepts of working with Spring Boot and the process that brought it to this point.

History and origin

To understand Spring, we need to understand the environment in which it emerged. When Java started making inroads into the server side, it introduced several backend APIs. These APIs were standardized by Sun Microsystems, but they were implemented by different Application Server vendors. These APIs presented a major step forward for the casual developer but included many limitations. Each vendor had a great deal of leeway in their implementation. All of them added their own unique APIs to differentiate their products further.

This created vendor lock-in despite the standardization efforts. The problem was that the standard did not cover many things. It was also very difficult to test in isolation with the old enterprise APIs. That was where Spring could deliver an advantage. The original version of Spring ran on top of Java Enterprise Edition, and it made the resulting apps vendor neutral and testable.

Spring was so successful it triggered a change in direction in the Java specifications to mimic Spring concepts. However, Spring kept moving faster and eclipsed the standard.

Spring Boot took the standard Spring Framework and packaged it with everything it needs, using a sophisticated build service and sensible defaults. This provides an “all in one” solution for typical backend applications. It made entry into Spring much easier, and it is the tool we will use today.

Inversion of Control, Dependency Injection, and Aspect Oriented Programming

There are the following three related concepts we need to understand when working with Spring:

1. **Inversion of Control (IoC)**
2. **Dependency Injection (DI)**
3. **Aspect Oriented Programming (AOP)**

These can be slightly confusing. However, they all relate to one another. Let us start with IoC. This is typical Java code one would write daily, as follows:

```
1. private final MyHelper myHelper = new MyHelper();
```

```
2.
3. public void myAPI() {
4. myHelper.doSomething();
5. }
```

There is nothing wrong with this code. It is good code. However, it limits your ability to do things. One of the big values in object-oriented programming is the ability to inherit and extend. We might want to extend **MyHelper** and verify that the call to **myAPI()** indeed triggered a call to **doSomething()**. That is a legitimate need. We might want to enhance the call in some way too, and to test that the user has permission to invoke **doSomething()**. We can do something like the following:

```
1. private final MyHelper myHelper = createMyHelper();
```

That code will create a different instance based on a situation, but then, we need to code all the logic of which instance would be returned in every situation in that method. IoC is about removing that noise. We normally control the exact object that will be assigned. With IoC, we invert that control and the framework takes over. It will decide the object to assign here. This code will work as the previous code did in Spring:

```
1. private final MyHelper myHelper;
2.
3. public MyClass(MyHelper myHelper) {
4. this.myHelper = myHelper;
5. }
```

It is the same as writing **new MyHelper()**. Spring will pass the right instance to the constructor. The same concept can also be expressed like the following, with the autowired annotation[3]:

```
1. @Autowired
2. private MyHelper myHelper;
```

Spring will make sure the field has the value you expect. When you run a test, it can mock it for you. It can even implement classes for you based on the interface definition. This makes it much easier to adapt your code and also reduces dependencies between the various parts of the application. Notice that this is dependency injection and IoC. IoC means that when we flip the control, we no longer explicitly allocate or set the value. DI supplies the implementation value where it is needed.

With IoC, we can break down a complex system into individual separate components (beans) that interact with one another through well-defined interfaces. However, they can be replaced dynamically. The two following

classes show a simple dependency:

```
1. public class A {
2. private final B dependency = new B();
3.
4. public void api() {
5. dependency.doSomething();
6. }
7. }
8.
9. public class B {
10. public void doSomething() {
11. // ...
12. }
13. }
```

This code is obvious and needs no further explanation. However, the following changes make this code into Spring code:

```
1. @Component
2. public class A {
3. private final B dependency;
4.
5. public A(B dependency) {
6. this.dependency = dependency;
7. }
8.
9. public void api() {
10. dependency.doSomething();
11. }
12. }
13.
14. @Component
15. public class B {
16. public void doSomething() {
17. // ...
18. }
19. }
```

Not much has changed. We remove the object allocation and use injection to add the dependency in Line 3. The other change is the **@Component** annotation. This annotation is a "stereotype". We will discuss stereotypes soon enough. What is important here is Line 10. In the previous listing, it was clear that **B.doSomething()** was invoked directly. Is this the case now?

Maybe a proxy was put in place in the injection stage, and all calls to **doSomething()** passed through that proxy. This is exactly what Spring does!

It uses proxies to provide elaborate functionality that we will explain soon enough. You will notice that the code is seamless. It is a standard method invocation, yet Spring can wrap functionality around every method and even replace its functionality seamlessly. This is AOP, with its ability to intervene and inject functionality surgically or globally. It is an amazing superpower in Spring, with one big caveat:

```
1. @Component
2. public class A {
3. private final B dependency;
4.
5. public A(B dependency) {
6. this.dependency = dependency;
7. }
8.
9. public void api() {
10. dependency.doSomething();
11. otherApi();
12. }
13.
14. public void otherApi() {
15. // ...
16. }
17. }
```

Notice the call to **otherApi()** at Line 11. Spring will not route it through the proxy, and any AOP functionality will not work. It would work if a different class invoked that API. The reason is obvious, we invoke the method on the current instance, and because of that, it will not trigger the proxy code created by Spring. This is a common pitfall for developers new to Spring.

Hello Spring Boot

Spring Boot made Spring development trivial. We no longer need a complex setup or configuration. By using a relatively simple template, we can create a self-contained application and even run it as any other Java application. Everything is packaged into a single "executable jar", as opposed to the overly complex configurations of the past.

To make things even simpler, the "Spring Initializr"[4] was created. We can use it to create a skeleton application with all the pieces and configuration options. *Figure 5.1* shows the initializr website configuration. We can use it to generate a simple hello world skeleton with a Maven build. It is highly configurable.

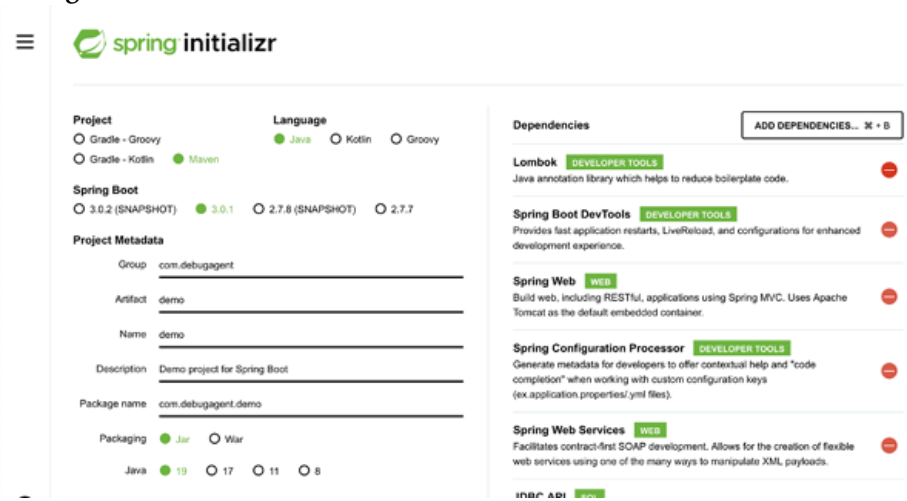

Figure 5.1: *Spring Initializr*

We can use the Spring Initializr via the website **https://start.spring.io** or via plugins to the various IDEs. There are a few things we need to configure for the app we build in this chapter:

- For project type: Maven.
- The latest version at the time of this writing is 3.0.1.
- **Group**: The package name for your application: "**com.debugagent**".
- **Java**: Latest model is Java 19.
- **Dependencies**:
    - Lombok
    - Spring Boot DevTools
    - Spring Web
    - Spring configuration processor

- ○ Spring Web services
- ○ JDBC API
- ○ Spring data JPA
- ○ HyperSQL database
- ○ Thymeleaf

Once you have picked and set all those options, we can generate the project. For the Web version, this will generate a downloadable ZIP file with the entire project in it. We can open this project in IntelliJ/IDEA and inspect it, and make sure to indicate that you trust the project when prompted. If you are using the IntelliJ/IDEA Ultimate Edition, then it includes built-in support for Spring Boot. This is the best option for professional developers; however, to keep the book accessible to everyone, we will still make use of the free community edition.

In *figure 5.2*, we see the hello world project created and opened in IntelliJ/IDEA Community Edition:

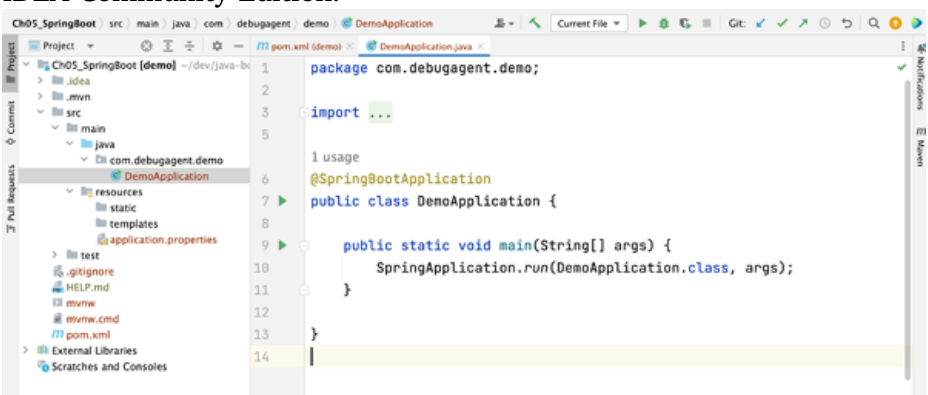

Figure 5.2: *Hello World Spring Boot project from the Initializr in IntelliJ/IDEA Community Edition*

You will notice the project is bare bones. Despite using a generator tool, the source tree has very little code. Where is that code?

Most of it is in the **pom.xml** file, which includes all the dependencies of the project. Spring Boot dependencies auto-configure themselves dynamically. We will start with something simple: a "proper" Hello Spring application.

In a Spring Boot application, it is important to maintain the package hierarchy. Spring Boot scans the packages under the root package by default. That means that if we created a package above the demo package, then Spring would not find our new classes. That is why we need to create a package under the **com.debugagent.demo** package. In this case, it will be **com.debugagent.demo.rest**. We will discuss REST soon enough, but

for now, we will create a class in this package:

```
1. package com.debugagent.demo.rest;
2.
3. import org.springframework.web.bind.annotation.GetMapping;
4. import org.springframework.web.bind.annotation.RestController;
5.
6. @RestController("/")
7. public class HelloWebService {
8. @GetMapping("/hello")
9. public String hello(String name) {
10. return "Hello " + name;
11. }
12. }
```

There are the following several things going on here:

- **RestController** is used to indicate that this class represents a RESTFull WebService. Notice we bind the class to the root URL.
- The method is annotated with **GetMapping**, which means that it will handle HTTP GET requests to the "/hello" URL.
- The API receives a parameter named "name".

What does all this mean? We can run this to find out. *Figure 5.3* shows the maven tool window in IntelliJ/IDEA. Notice the expanded section that points at **spring-boot:run**.

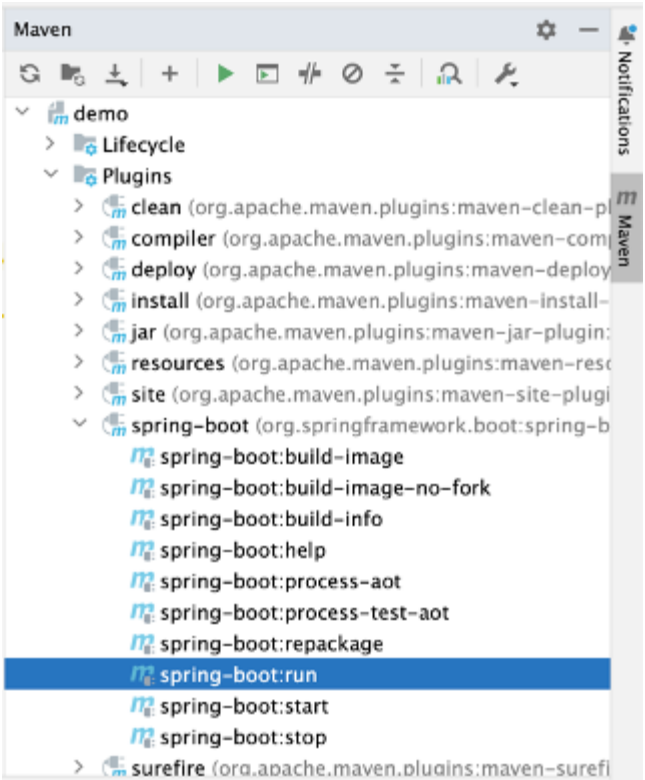

Figure 5.3: *Spring-Boot:run option in the IntelliJ/IDEA Maven Tool Window*

When we right-click that option, as shown in *figure 5.4*, we are presented with the option to run this target. Once we do that, Spring Boot will launch.

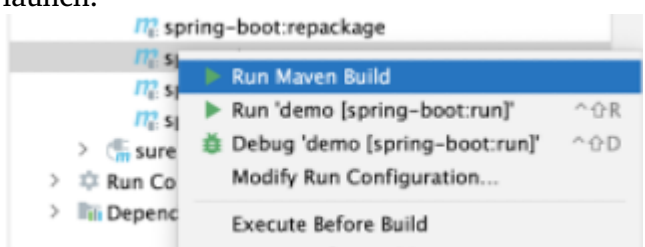

Figure 5.4: *Run and debug options for running a Spring Boot target*

Progress should show in the console, and once the project runs, Spring will print: “Completed initialization”. Now we can test the code by visiting **http://localhost:8080/hello?name=Shai** in the browser, which will print out “Hello Shai”. You can similarly use the **curl** command line tool or Postman to achieve the same effect.

What we did here is to create a simple Web service that works over HTTP without writing any complex code. We implemented a trivial method, and

everything else was decided by convention. That is the essence of Spring Boot. It handles everything for us and tries to make smart “common” choices.

Spring is much more than a Web solution. Let us say that we have a REST API, a Web Front End, and a Mobile Front End. All three front ends need similar but subtly different types of solutions. The common law in programming is **Don’t Repeat Yourself** (**DRY**); we need to write the code only once. However, we will need custom versions for each front end.

The solution is simple; we have multiple front-end controllers and they all use a single backend service. This will let us keep the code simple yet perform specific customizations per client. To do this, we can create a service package next to the rest package and add the following class:

```
1. @Service
2. public class HelloService {
3. public String helloService(String name) {
4. return "Hello " + name;
5. }
6. }
```

Notice that a service is even simpler than a **RestController**. We only include business logic and nothing else. Now, we can inject the service into the **RestController**; the resulting code seems longer and more complex, but this is only because the demo is so trivial:

```
1. @RestController
2. public class HelloWebService {
3. private final HelloService helloService;
4.
5. public HelloWebService(HelloService helloService) {
6. this.helloService = helloService;
7. }
8.
9. @GetMapping("/hello")
10. public String hello(String name) {
11. return helloService.helloService(name);
12. }
13. }
```

This works reasonably well when we have one object injected, but adding multiple objects to the constructor is painful. Luckily, we added a library

called Lombok to the set of libraries in Spring. Lombok replaces Java boilerplate with annotations. It requires an IDE plugin to work properly, and once installed, we can replace the previous listing with the following shorter code:

```
1. @RestController
2. @RequiredArgsConstructor
3. public class HelloWebService {
4. private final HelloService helloService;
5.
6. @GetMapping("/hello")
7. public String hello(String name) {
8. return helloService.helloService(name);
9. }
10. }
```

Notice that this code will not compile without Lombok. We have a final field that we do not initialize. This is illegal in Java. However, the **@RequiredArgsConstructor** makes sure that a constructor is generated for the class. That means that writing **new HelloWebService(helloService)** will work as expected. For a single field, this is a small impact, but as our code grows, it can become a major advantage. As you will soon see, Lombok has many other powers as well.

You might recall that when we looked at the first sample of a Spring Boot class, we used the **@Component** annotation. We then used a **@Service** to denote a “bean”; these are both stereotypes. A stereotype marks the object as one that is managed by Spring. When a Spring application loads, it scans the package hierarchy under the application for “beans”. It then inspects those beans and generates code, proxies, and so on to implement the declared functionality.

There are four standard stereotype annotations: Component, Repository, Service, and Controller. Notice that this list is missing annotations such as RestConroller, and that is because it is an alias for Controller. We pick the right stereotype based on the specific requirements of a class.

A REST API

The REST code we wrote up until this moment is relatively simple. However, REST APIs can be sophisticated. They can include objects and details that we might need to keep track of. We will start with a brief overview of REST.

REpresentational State Transfer (**REST**) is an architectural style for

building Web services. It is based on a client-server model, where the client sends a request to the server, and the server returns a response. REST defines a set of constraints to be followed when creating Web services, such as being stateless and using a uniform interface. This allows for flexibility and scalability in the way that the service is implemented and consumed. REST is often used for building Web services that are consumed by Web and mobile applications.

The core principles behind REST are as follows:

- **Client-Server**: The separation of the client and server allows for a separation of concerns and allows for the evolution of the two independently.
- **Statelessness**: The server does not store any information about the client's state, meaning that each request from the client must contain all the information necessary for the server to understand and fulfill the request.
- **Cacheability**: Responses from the server can be cached by the client, which can improve performance by reducing the number of requests that need to be made.
- **Layered system**: The architecture is composed of multiple layers, with each layer serving a specific purpose. This allows for flexibility and scalability in the way that the service is implemented and consumed.
- **Code on demand (optional)**: Servers can temporarily extend or customize the functionality of a client by sending executable code.
- **Uniform interface**: All resources share the same interface, which consists of a small set of well-defined methods (such as GET, POST, PUT, and DELETE). This allows for a consistent and simple way to interact with resources and allows for the easy creation of generic client libraries.

What this typically means in practice is that the client uses an HTTP request and receives the response as JSON. But so far, we returned a String as a response. That is not proper REST.

To demonstrate this, we will create a simple exercise Web service that will send a JSON response. To start, we need to define the JSON structure. In this case, JSON maps rather nicely to a class, but we do not need to go that far. A Java record can do the trick:

```
1. public record ExerciseDTO(String question, List<String> incorrectAnswers,
   String correctAnswer) {
2. }
```

A **DTO** is a shorthand for **Data Transfer Object**. This is a common pattern in which we use an object representation to transfer data between tiers. This saves us the hassle of creating multiple APIs for each of the entries, and we can perform requests as a single operation with a DTO.

Next, we need to create an API that will return the JSON data. First, we will create the service as a simple mock implementation:

```
1. @Service
2. public class ExerciseService {
3. private static final ExerciseDTO[] EXERCISES = {
4. new ExerciseDTO("1/2 * 1/2 =", List.of("2/2", "1/2", "1/8"), "1/4"),
5. new ExerciseDTO("1/2 + 1/2 =", List.of("2/2", "1/2", "1/8"), "1"),
6. new ExerciseDTO("1/2 : 1/2 =", List.of("2/2", "1/2", "1/8"), "1"),
7. new ExerciseDTO("1/2 - 1/2 =", List.of("0/2", "-1/2", "1/0"), "0")
8. };
9. private final Random RANDOM = new Random();
10.
11. public ExerciseDTO getExercise() {
12. return EXERCISES[RANDOM.nextInt(EXERCISES.length)];
13. }
14. }
```

Finally, we can create a Web service that will use that service:

```
1. @RestController
2. @RequiredArgsConstructor
3. public class ExerciseWebService {
4. private final ExerciseService exerciseService;
5.
6. @GetMapping("/getExercise")
7. public ExerciseDTO getExercise() {
8. return exerciseService.getExercise();
9. }
10. }
```

Notice that we return the record directly. So where is the JSON that was promised?

With these changes, we can request the URL **http://localhost:8080/getExercise**. This returns the following result (formatted for your

convenience):

```
1. {
2. "question":"1/2 + 1/2 =",
3. "incorrectAnswers":[
4. "2/2",
5. "1/2",
6. "1/8"],
7. "correctAnswer":"1"
8. }
```

Spring Boot converted the record we returned and the list within it to JSON. This is the default behavior for Spring Boot and can be overridden to produce XML or other sophisticated behaviors.

What if we want to accept question suggestions? It is just as simple. We need to add the following method to the Web service:

```
1. @PostMapping("/submitExercise")
2. public void submitExercise(@RequestBody ExerciseDTO exerciseDTO) {
3. exerciseService.submitExercise(exerciseDTO);
4. }
```

The one interesting element here is the **RequestBody** annotation, which we use to indicate how we will receive the argument. Next, we need to add the following code to the service class:

```
1. private final List<ExerciseDTO> userSuggestions = new ArrayList<>();
2.
3. public void submitExercise(ExerciseDTO exerciseDTO) {
4. userSuggestions.add(exerciseDTO);
5. }
```

Submitting a POST request through the browser is more challenging. We can use **cURL** to submit the following request:

```
curl -X POST  -H "Content-Type: application/json" -d '{"question":"1/3
+ 1/3 =","incorrectAnswers":["2/6", "1/6", "1"],"correctAnswer":"2/3"}'
"http://localhost:8080/submitExercise"
```

Notice that the command accepts the body of the JSON, and the content type header is set to JSON. Without that header, the server would not know how to process the input.

Spring MVC and Thymeleaf

REST is great, but we might want to create a Web interface and not just a Web service. One way we can do that is through Spring MVC. Spring MVC is built on the **Model-View-Controller** (**MVC**) design pattern, which separates the application logic into three distinct components: the model, the view, and the controller. The **model** represents the data and business logic of the application, the **view** is responsible for displaying the data to the user, and the **controller** handles incoming requests and coordinates the interaction between the model and the view.

Spring MVC provides a flexible and extensible architecture for building Web applications, making it a popular choice for both small and large-scale projects. In *figure 5.5,* we can see a diagram of the classic MVC pattern.

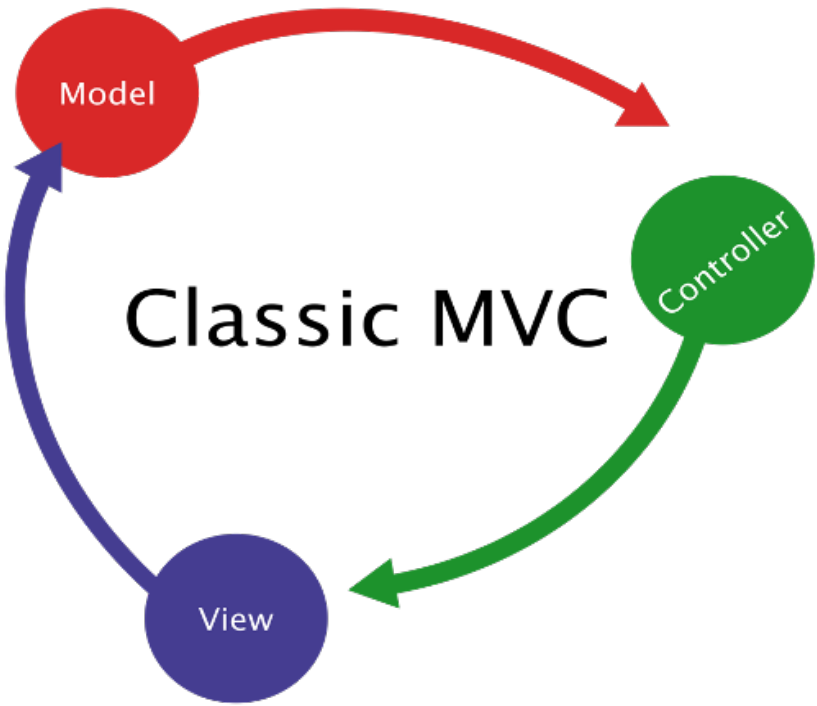

Figure 5.5: *The classic MVC pattern*

The Spring MVC version includes all three parts of the MVC puzzle, but they are arranged a bit differently, as seen in *figure 5.6.*

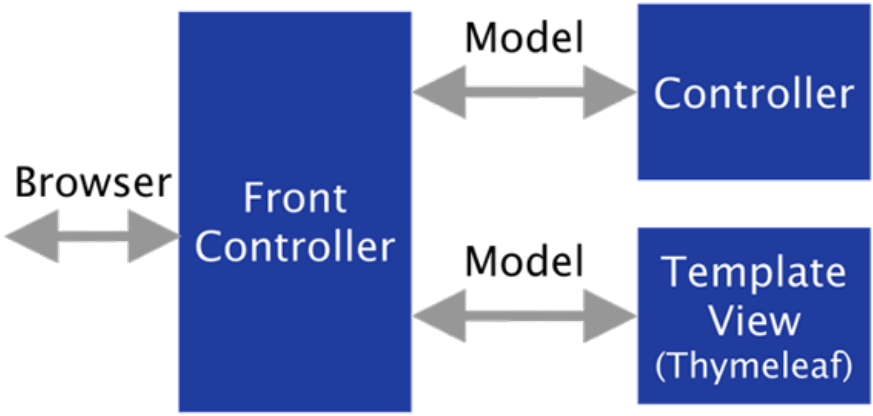

Figure 5.6: *Spring MVC basic structure*

The view portion of Spring MVC is pluggable and can be implemented with any common templating engine. One of the popular choices is Thymeleaf. Thymeleaf is a Java-based template engine that is used to process and render HTML, XML, JavaScript, and other markup languages. It is designed to work seamlessly with the Spring Framework but can also be used in standalone applications. Thymeleaf's templates are written in standard XHTML/HTML5 and use special attributes and tags to bind data to the template. This allows for a separation of concerns between the presentation and the data, thus making the code more maintainable. Templating engines are tools that are used to generate dynamic HTML pages. They allow developers to define a template that includes placeholders for dynamic content and then fill in the placeholders with actual data at runtime. This can make it easier to create and maintain consistent layouts across multiple pages of a website or application.

Thymeleaf has several benefits over other templating engines:

- **Natural templating**: Thymeleaf templates are written in standard XHTML/HTML5, making them easy to read and understand for developers who are already familiar with these markup languages. This also makes it easy for designers to work with the templates, as they are already familiar with the syntax.
- **Spring framework integration**: Thymeleaf is designed to work seamlessly with the Spring framework for more seamless integration of the template engine into the overall application.
- **Validation and type-safety**: Thymeleaf templates are parsed and validated at the template level, which can prevent errors from occurring at runtime. This means that developers can catch errors early on, which can then save time and reduce the number of bugs

in the application.

- **Internationalization (i18n) support**: Thymeleaf has built-in support for internationalization, which allows developers to create templates that can be easily localized for different languages and cultures.
- **Flexibility**: Thymeleaf can be used in both Web and non-Web environments, making it more versatile than some other template engines that are designed for Web use only.
- **Good performance**: Thymeleaf is designed for high-performance, which makes it suitable for high-traffic websites and Web applications.

A simple “Hello World” Thymeleaf template would look as shown as follows:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<title>Hello World</title>
</head>
<body>
<h1 th:text="'Hello, World!'">Hello, World!</h1>
</body>
</html>
```

In this example, the template is written in standard XHTML/HTML5, with the addition of the Thymeleaf-specific attribute **th:text** on the <**h1**> tag.

The **th:text** attribute is used to bind the value “Hello, World!” to the text of the <**h1**> tag, which will be rendered as “Hello, World!” in the browser.

The **xmlns:th** attribute on the <**html**> tag is used to define the Thymeleaf namespace, which allows the use of Thymeleaf-specific attributes and tags in the template.

When this template is processed by Thymeleaf, it will generate the following output:

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>
</head>
```

```
6. <body>
7. <h1>Hello, World!</h1>
8. </body>
9. </html>
```

As you can see, the **th:text** attribute is removed, and the value “Hello, World!” is displayed in the browser. This example is a very basic template and does not use any dynamic data. However, it illustrates the basic concepts of how Thymeleaf binds data to a template.

We can integrate this with Spring MVC, using the following template to display a question. It is important to place this template under the **src/main/resources/templates** directory so that Spring can treat it as a template. In this case, we need to give it the name **template.html**, which we will use soon:

```
1. <!DOCTYPE html>
2. <html lang="en" xmlns="http://www.w3.org/1999/html">
3. <head>
4. <meta charset="UTF-8">
5. <title>Hello ThymeLeaf</title>
6. </head>
7. <body>
8. <form method="post" action="/answer">
9. <fieldset>
10. <legend th:text="${question}">This is the question...</legend>
11.
12. <div th:each="answer : ${answers}">
13. <input type="radio" th:id="${answer}" name="correctAnswer" th:value="${answer}">
14. <label th:for="${answer}" th:text="${answer}">Answer</label>
15. </div>
16. <input type="hidden" name="questionId" th:value="${questionId}">
17. <input type="submit">
18. </fieldset>
19. </form>
20. </body>
21. </html>
```

In Line 10, we have a dummy question. The value within the legend tag will be replaced by the value of the question attribute in the spring model.

Notice the syntax with the $ symbol and the curly brackets. When we define the model, we will set a value to the question and to the list of answers. These values will then be displayed in the template.

In Line 12, we can see a for-each loop. The value of ${answers} is a list that we iterate, assigning each entry to a ${answer}. Assuming the list has four elements, we will have four copies of the div element and its content with values matching each of the entries. That is a very powerful tool for the creation of a Web UI.

The following input tags represent the values we wish to submit when sending the form to the "/answer" URI, which will test the response to the question.

Next, we need to inspect the controller part of the application:

```
1. @Controller
2. @RequiredArgsConstructor
3. public class WebUI {
4. private final ExerciseService exerciseService;
5.
6. @GetMapping("/question")
7. public String question(Model model) {
8. ExerciseDTO exerciseDTO = exerciseService.getExercise();
9. model.addAttribute(«question», exerciseDTO.question());
10. var answers = new ArrayList< >(exerciseDTO.incorrectAnswers());
11. answers.add(exerciseDTO.correctAnswer());
12. Collections.shuffle(answers);
13. model.addAttribute("answers", answers);
14. model.addAttribute("questionId", exerciseDTO.questionId());
15. return "template";
16. }
17. }
```

We inject the exercise service into the class using Spring and Lombok (for the constructor injection). This lets us reuse the functionality we wrote earlier.

The question call accepts a model object which is injected seamlessly by Spring MVC. This model lets us pass values to the templates. The method returns a string which is the name of the template file, without the html suffix. This maps to the **template.html** file we previously created.

We set the attribute values of the question, answers, and questionId. Once we do all of that, we can visit **http://localhost:8080/question** and see something like *figure 5.7*.

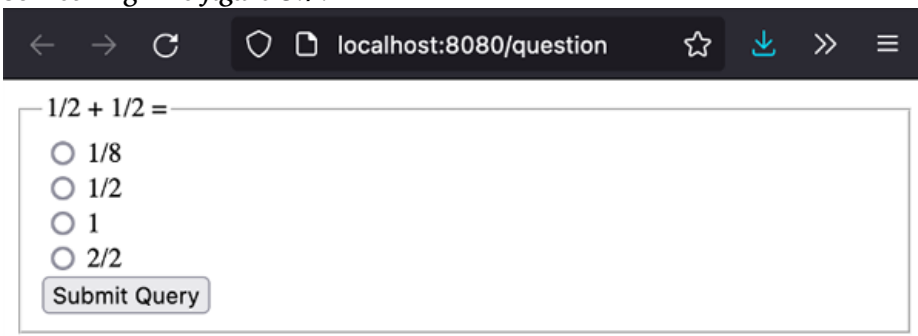

Figure 5.7: *Initial template*

This user interface is of no use at this point since we cannot submit it.

Notice that to know which question we will address in the UI, we needed a question ID field. To determine that, we should add that ID to the ExerciseDTO record as such:

```
1. public record ExerciseDTO(long questionId,
2. String question,
3. List<String> incorrectAnswers,
4. String correctAnswer) {
5. }
```

We need to also update the constant containing the default ExerciseDTO objects:

```
1. private static final ExerciseDTO[] EXERCISES = {
2. new ExerciseDTO(1,"1/2 * 1/2 =", List.of("2/2", "1/2", "1/8"), "1/4"),
3. new ExerciseDTO(2,"1/2 + 1/2 =", List.of("2/2", "1/2", "1/8"), "1"),
4. new ExerciseDTO(3,"1/2 : 1/2 =", List.of("2/2", "1/2", "1/8"), "1"),
5. new ExerciseDTO(4,"1/2 - 1/2 =", List.of("0/2", "-1/2", "1/0"), "0")
6. };
```

The next step is handling the submission of an answer, which we can do by adding the following code to the WebUI class:

```
1. @PostMapping("/answer")
2. public String answer(@ModelAttribute ExerciseDTO body) {
3. Long id = body.questionId();
4. ExerciseDTO exerciseDTO = exerciseService.getExerciseById(id);
```

```
5. if(body.correctAnswer().equals(exerciseDTO.correctAnswer())) {
6. return "correct";
7. }
8. return "incorrect";
9. }
```

There are several things we need to notice about this code. We pass the variables for the model as a body object. In this case, we reuse the record object, but it can be any arbitrary object type. We return two different values: correct and incorrect, which we will get to soon enough.

However, the more important part is the **getExerciseById(long)** method. We need to add this API to find the right exercise to compare against. We could theoretically use a session object to keep track of the current state. But there are some drawbacks to that as well, which are explained as follows:

- **Increased memory usage**: Storing large amounts of data in the session can cause the server's memory to become saturated, leading to performance issues.
- **Security risks**: Storing sensitive information in the session can make it vulnerable to attacks, such as session hijacking or tampering.
- **Stateless nature of HTTP**: The session object is based on a client-side cookie, which can be deleted or blocked by the client. This can cause issues with maintaining state across requests. It can further make things difficult with regulations related to cookies, such as GDPR.
- **Scaling issues**: The session object is typically stored in memory on the server, which can make it difficult to scale horizontally.
- **Difficulty in testing**: Because the session object is based on a client-side cookie, it can be difficult to test the application without a live Web browser.

To avoid these problems, we pass the question ID and the answer when submitting the form. Thus, all the data we need is within the request.

To do that, we need a way to find a question by its ID, which we can do by adding this method to **ExerciseService.java**:

```
1. public ExerciseDTO getExerciseById(long id) {
2. return Arrays
3. .asList(EXERCISES)
4. .stream()
```

```
5. .filter(e -> e.questionId() == id)
6. .findFirst()
7. .orElseThrow();
8. }
```

This method seems somewhat complex. However, all the ideas within it are ideas we covered before:

- The **asList()** method presents an array as if it was a **List** object.
- This lets us use the **stream()** method and the functional expression on it.
- The Stream API includes the filter option which we use to filter in the entry with the right question ID.
- We only need one entry which is the reason we use **findFirst()**. This returns an **Optional<ExerciseDTO>** instance.
- Optional is a Java class that solves the problem of nullability in functional programming. In this case, if the value is missing, the code will throw a **RuntimeException**. Otherwise, the value will be returned.

The **Optional** class is a container object which may or may not contain a non-null value. It is used to represent a value that may or may not be present to avoid null pointer exceptions. It can be used as an alternative to null checks and provide a more elegant way to handle null values. It has various methods for working with the value it contains, such as **isPresent()** to check if a value is present, **get()** to retrieve the value, or **orElse()** to provide a default value if the Optional is empty.

Stream APIs fail badly when working with null values, as we often chain API calls to one another. If a null value is in the middle, the failure can cascade. Optional provides an API that should never be null; it can be empty, at which point we can provide a default value.

Finally, when the answer call succeeds or fails, we need two templates to match each state. This is **correct.html**:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Correct Answer</title>
6. </head>
7. <body>
8. <h1>Well done!</h1>
```

```
9. <p><a href="/question">/Try Again</a>?</p>
10. </body>
11. </html>
```

The following is **incorrect.html**:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Incorrect Answer</title>
6. </head>
7. <body>
8. <h1>Sorry, that isn't correct...</h1>
9. <p><a href="/question">/Try Again</a>?</p>
10. </body>
11. </html>
```

With these changes, the application will work as expected and let us pick the right or wrong answer. Still, when we visit **http://localhost:8080/**, we will get an error page. How do we handle the root URI?

We can use MVC for that, although there exists a much simpler way. We can use the static directory under **src/resources/static** to store static files for the project. Typically, we would use that directory for CSS and image files that have no template logic. We can also store plain HTML files in that directory so that we will have a decent welcome message when visiting the server. If we use the name **index.html**, it will be served for the server root implicitly:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Welcome Page</title>
6. </head>
7. <body>
8. <h1>Hello Spring</h1>
9. <p>This is a standard static page in Spring Boot.
10. We can serve static resources by placing them in
11. the static directory.</p>
12. <p>Get a dynamic exercise by going to
```

```
13. <a href="/question">/question</a>.</p>
14. </body>
15. </html>
```

SQL and JDBC

Up until this point, we kept information in memory. We need to store the current application state in a permanent store. In typical systems, this is a database, and in most cases, this is still a relational SQL database. There is a lot of theory around databases, and there are many database types.

SQL basics

Structured Query Language (**SQL**) is the prevailing standard in the world of relational databases. There are many important books on the matter; one should understand the concepts behind relational databases before embarking on them. Still, we will try to give a brief overview here to get started.

A relational database is a type of database that stores data in the form of tables, with each table consisting of a set of rows and columns. The rows represent individual records, whereas the columns represent the attributes of those records. The tables are related to one another through the use of keys, which are used to link data from one table to another. The most common type of key is the primary key, which is unique for each record in a table, and is used to identify that record. Foreign keys are used to link data from one table to another, creating a relationship between the tables. The primary goal of a relational database is to organize data in a way that is easy to understand and to make it easy to query and retrieve data from the database.

Another important concept in databases is transactions. A transaction is a sequence of one or more database operations that are executed as a single atomic unit of work. The operations within a transaction are typically executed in a specific order and are designed to maintain the consistency and integrity of the data in the database.

Transactions typically include one or more of the following operations:

- **Create**: Adding new data to the database.
- **Read**: Retrieving data from the database.
- **Update**: Modifying existing data in the database.
- **Delete**: Removing data from the database.

This forms the acronym: CRUD.

A transaction can also include multiple operations of the same type, for example, multiple updates. Transactions can be used to perform complex tasks that involve multiple operations, such as transferring money from one bank account to another, where one account is debited, and the other is credited. In this case, the transaction would include two operations, one to debit one account and the other to credit the other.

Transactions are important because they provide a mechanism for ensuring that the data in the database remains consistent and accurate, even in the event of system failures or errors. This is achieved by the **ACID** properties, which is basically an acronym for **Atomicity**, **Consistency**, **Isolation, and Durability**.

A key feature of a transaction is that it can be committed or rolled back. Once a transaction is committed, its changes are made permanent in the database. If a transaction is rolled back, its changes are undone, and the database is returned to its.

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These are a set of properties that ensure that the database transactions are processed reliably. Let us now know each aspect in detail:

- **Atomicity**: This property ensures that a transaction is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction is rolled back to its previous state.
- **Consistency**: This property ensures that a transaction brings the database from one valid state to another. All data integrity constraints and rules are enforced during a transaction, and the database remains in a consistent state before and after the transaction is executed.
- **Isolation**: This property ensures that concurrent transactions do not interfere with one another. Each transaction is executed in isolation, and the changes made by one transaction are not visible to other transactions until the first transaction is committed.
- **Durability**: This property ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures.

Together, these properties ensure that database transactions are processed reliably, with minimal risk of data corruption or loss. Notice that ACID is rarely “perfect”. Working with a database is a delicate balance of performance versus proper isolation. A fully isolated database would prevent concurrent work completely and become a single-user system.

We will start with the hypersonic database, which we added to the Spring Boot starter project. The main reason for that is that it requires no setup or

configuration.

Java Database Connectivity

Java Database Connectivity (**JDBC**) is a Java API that provides a standard way for Java programs to interact with relational databases. It allows Java developers to connect to a wide variety of databases, execute SQL statements, and process the results. It is a low-level API that lets us use SQL directly from Java and process the results.

In most modern projects, we use an abstraction layer such as **Java Persistence Architecture** (**JPA**), which we will discuss later in the chapter. However, such abstractions are considered leaky abstractions. They expose details of the underlying systems and could fail if we have a limited understanding of those underlying systems. That is why it is important that we understand JDBC and SQL, even if we will always work through JPA.

JDBC is a challenging API. In JDBC, it is important to clean up connections and resources when we are done with them. Failure to do proper cleanup can result in hard-to-detect failures over time. Thankfully, the JDBC API adopted AutoClosable in recent versions and can use the try-with-resource support. Spring makes this even simpler by managing and pooling database connections. As with threads, it is common practice to pool database connections to conserve resources.

One of the biggest problems with SQL is portability. This is best demonstrated by a simple example. If we want to take the exercise code from before and convert it to an SQL database, we will start by creating two tables: **EXERCISES** and **ANSWERS**. The tables would have a one-to-many relation, whereas the answers table would have a foreign key to the exercise table. This means that for every row in the exercises table, we would have multiple answers in the answers table. We can then perform a “join” to query the data. This is the “bread and butter” of SQL development, and it is very basic SQL. To use SQL within Spring, we can use code like the following:

```
1. @Service
2. @RequiredArgsConstructor
3. @Transactional
4. public class DBAccess {
5. private final DataSource dataSource;
6.
7. @PostConstruct
8. public void initDB() {
```

```
9. JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
10. ....
11. }
12. }
```

There are several interesting things in this code. First and foremost is the Transactional annotation on the class. Spring manages transactions for us using the proxy classes. When a method enters, it is the equivalent of starting a new transaction. If a bean invokes another transactional bean, the transaction continues as a single unit. This is called “declarative transactions” and with it, we can control refined behaviors such as isolation levels, origin transactional state, and more.

We inject the **DataSource** object, which handles the database connections. We use that to create a **JdbcTemplate**, which we can use to execute queries. Notice the method where this is all defined is the **PostConstruct** method. This is like writing code in a constructor. However, unlike constructors, the method is invoked after injection is completed. Had we written this code in the constructor, the **DataSource** might have still been null.

Next, we create two SQL tables. Notice we use the new syntax for multiline strings in Java:

```
1. jdbcTemplate.execute(
2. """
3. CREATE TABLE EXERCISES (
4. id INT PRIMARY KEY IDENTITY,
5. question VARCHAR(255),
6. correctAnswer VARCHAR(255))
7. «"");
8. jdbcTemplate.execute(
9. """
10. CREATE TABLE ANSWERS (
11. id INT PRIMARY KEY IDENTITY,
12. exercise_id INT NOT NULL,
13. answer VARCHAR(255),
14. FOREIGN KEY (exercise_id) REFERENCES EXERCISES(id))
15. «"");
```

The first table is the standard exercises table. Lines 4 and 11 include the primary keys for the tables. In both cases, it is a number that is generated

by the database for us. This syntax is database specific in many cases. Next, we have two Strings of up to 255 characters. This is standard SQL.

The interesting bit is in Line 14, where we define the relation between the answers table and the exercises table. This means that we cannot add a row to the answers table without setting **exercise_id**, so it will point at the right database row.

We can try to add something to the table. Adding an exercise seems trivial enough:

```
1. jdbcTemplate.update("INSERT INTO EXERCISES (question, correctAnswer)
   values ('1/2 * 1/2 =', '1/4')");
```

This adds an exercise, but this code has a problem. We want to add variables and not values. Hard coding the value in the SQL is unhelpful. Something like the following might be more useful:

```
1. jdbcTemplate.update("INSERT INTO EXERCISES (question, correctAnswer)
   values ('" + question +"', '" + answer + "')");
```

This is a big security vulnerability called SQL injection. A malicious hacker can send an SQL statement as a question or answer, and this can give the hacker full access to your system. You must never use code like that.

Instead, we should use prepared statements as such:

```
1. jdbcTemplate.update("INSERT INTO EXERCISES (question, correctAnswer)
   values (?, ?)",
2. "1/2 * 1/2 =", "1/4");
```

Even if the arguments at Line 2 would be replaced with SQL, JDBC would escape it, and the SQL would get written to the database. The syntax is also much nicer.

We now added a row with an exercise but what about the questions? This is where we fail. The syntax should be as follows:

```
1. INSERT INTO ANSWERS (exercise_id, answer) values (?, ?)
```

This seems reasonable enough. But what is the value of **exercise_id**?

We need to get a result from the previous insert statement. This is possible, but it is database specific. This is the official Spring way of doing this. It is a reasonably cross-platform, but it does not work for the database that we are using:

```
1. KeyHolder keyHolder = new GeneratedKeyHolder();
2. jdbcTemplate.update(con -> {
3. var preparedStatement =
```

```
4. con.prepareStatement("INSERT INTO EXERCISES (question, correctAnswer) values (?, ?)");
5. preparedStatement.setString(1, "1/2 * 1/2 =");
6. preparedStatement.setString(2, "1/4");
7. return preparedStatement;
8. }, keyHolder);
9. var key = keyHolder.getKey();
10. String query = "INSERT INTO ANSWERS (exercise_id, answer) values (?, ?)";
11. jdbcTemplate.update(query, key, "2/2");
```

Here we use a prepared statement and keyholder to get the value of the generated key, which we then pass into the answers. Unfortunately, the value of the key is null in the database of our choice.

There are solutions to these problems. But as you can see, this is getting very cumbersome. We even skipped the discussion of database creation, which should be a one-time thing (also not very portable). Note that typically one would use a database migration tool for setup and creation, such as Flyway, Liquibase, and so on.

Java Persistence Architecture

As the previous section made abundantly clear, SQL is difficult. The basics are simple enough, but we quickly run into rough edges. This is especially true if we wish to remain vendor agnostic. That is where **Java Persistence Architecture** (**JPA**) comes in. It is a standard that was formed due to the success of the **Hibernate ORM**, which is still one of the most popular JPA implementations.

An **Object Relational Mapper** (**ORM**) is an API that converts objects from an object-oriented language (such as Java) to a relational model (like a database) and vice versa. The history of object-oriented programming is littered with such tools; Hibernate is far from the first such tool. However, it was innovative in its simplicity and provided one “killer feature”: it let developers access SQL.

Some ORM tools try to hide the underlying database completely. They aim for seamlessness as they load and save objects from storage. For them, the SQL is just an “implementation detail”. Vendors promoted them as tools that would rid you of the need to learn SQL.

Hibernate bucked that trend. It lets developers write SQL directly. It was designed with SQL developers in mind. We still need to understand SQL to use Hibernate, but we can write more portable code, leverage caching, and work with objects. This became a popular notion, and today, JPA is the

leading standard by a good margin.

In JPA, we work with Entity objects that can be mapped to tables, views, and so on. Spring has its own abstraction on top of JPA: Spring Data. It offers many features, one of which is the CRUD repository, that we can use to manage entities. To get started, let us look at the answer entity:

```
1. @Entity
2. @Getter
3. @Setter
4. public class Answer {
5. @Id
6. @GeneratedValue(strategy = GenerationType.IDENTITY)
7. private Long id;
8.
9. private String answer;
10.
11. public Answer() {
12. }
13.
14. public Answer(String answer) {
15. this.answer = answer;
16. }
17. }
```

There are many things going on here. First, we have the Entity annotation, which we need to declare a JPA entity. Next, we declare the getters and setters. All entities in JPA must be Java Beans, which means that the fields within the class are used with getters and setters. Unfortunately, since records are immutable, they cannot be used as JPA entities. The getter and setter annotations are a part of Lombok. A common mistake that developers often make is using the **@Data** annotation, which includes both the getters and the setters. However, that annotation also defines the **equals()** and **hashcode()** methods of the class, and these can cause serious problems. As an entity is modified, it might have an invalid state for a while, which might leave its identity in flux and might create a problem with JPA. That is why we should be very careful when implementing **hashcode()** in JPA.

The **id** field is the primary key for the entity. It is numeric for performance and simplicity. In Line 6, we indicate that we want the id to be generated for us using the identity strategy. There are several generation strategies,

such as **Universal Unique Identifier** (**UUID**) and similar strategies. Finally, notice that a JPA entity must have a default constructor. We added the other constructor for convenience.

The next entity is very similar:

```
1. @Entity
2. @Getter
3. @Setter
4. public class Exercise {
5. @Id
6. @GeneratedValue(strategy= GenerationType.IDENTITY)
7. private Long id;
8.
9. private String question;
10. private String correctAnswer;
11.
12. @OneToMany
13. private Set<Answer> answers;
14.
15. public Exercise() {
16. }
17.
18. public Exercise(ExerciseDTO exerciseDTO) {
19. question = exerciseDTO.question();
20. correctAnswer = exerciseDTO.correctAnswer();
21. }
22.
23. public ExerciseDTO getDTO() {
24. return new ExerciseDTO(id, question,
25. answers
26. .stream()
27. .map(Answer::getAnswer)
28. .collect(Collectors.toList()),
29. correctAnswer);
30. }
31. }
```

The important part here is the OneToMany mapping between the answer entity and this entity. By default, JPA relations are eager, which is problematic. However, that opens a big discussion that should belong in a JPA book. The additional constructor and getter methods are there for convenience.

As we discussed before, Spring Data includes a CRUD repository feature which we can leverage by defining an interface as such:

```
1. @Repository
2. public interface ExerciseRepository extends JpaRepository<Exercise, Long>
   {
3. }
4.
5. @Repository
6. public interface AnswerRepository extends JpaRepository<Answer, Long> {
7. }
```

The interface is blank, but we can add methods by following a common naming convention. For example, we can add the following method to the exercise repository:

```
1. List<Exercise> findByQuestionIgnoreCase(String question);
```

It will be implemented implicitly and will work exactly as you would expect. The repository itself includes multiple methods such as **count()**, **findAll()**, and so on. We can also define additional methods and write the SQL (or JQL, which is JPA's SQL syntax) to run a specific query.

The repositories are implemented by Spring, and no code is necessary. We simply inject the repository into a bean and start using it. The following code adds all the hardcoded exercises into the database:

```
1. @PostConstruct
2. public void init() {
3. for(ExerciseDTO exerciseDTO : EXERCISES) {
4. saveExercise(exerciseDTO);
5. }
6. }
7.
8. private void saveExercise(ExerciseDTO exerciseDTO) {
9. Exercise exercise = new Exercise(exerciseDTO);
10. Set<Answer> answers = new HashSet<>();
11. for(String answer : exerciseDTO.incorrectAnswers()) {
```

```
12. answers.add(new Answer(answer));
13. }
14. answerRepository.saveAll(answers);
15. exercise.setAnswers(answers);
16. exerciseRepository.save(exercise);
17. }
18.
19. public ExerciseDTO getExercise() {
20. return exerciseRepository.findAll()
21. .get(RANDOM.nextInt((int)exerciseRepository.count())).getDTO();
22. }
23.
24. public ExerciseDTO getExerciseById(long id) {
25. return exerciseRepository.findById(id)
26. .orElseThrow()
27. .getDTO();
28. }
29.
30. public void submitExercise(ExerciseDTO exerciseDTO) {
31. saveExercise(exerciseDTO);
32. }
```

Spring **application.properties** lets us set various values, including forcing verbosity in JPA using **spring.jpa.show-sql = true**. Once we turn that on, we can inspect the output of the JPA code that imports the exercises to the database (output was trimmed):

```
1. Hibernate: drop table if exists answer cascade
2. Hibernate: drop table if exists exercise cascade
3. Hibernate: drop table if exists exercise_answers cascade
4. Hibernate: create table answer (id bigint generated by default as identity (start with 1), answer varchar(255), primary key (id))
5. Hibernate: create table exercise (id bigint generated by default as identity (start with 1), correct_answer varchar(255), question varchar(255), primary key (id))
6. Hibernate: create table exercise_answers (exercise_id bigint not null, answers_id bigint not null, primary key (exercise_id, answers_id))
7. Hibernate: alter table exercise_answers add constraint UK_ul67q9o6skfxjjepuila2kmv unique (answers_id)
8. Hibernate: alter table exercise_answers add constraint
```

FK1jwaqby0k37hj8l5licma2x50 foreign key (answers_id) references answer

9. Hibernate: alter table exercise_answers add constraint FKec42hfpshagny1o9v597as4p2 foreign key (exercise_id) references exercise
10. Hibernate: insert into answer (id, answer) values (default, ?)
11. Hibernate: insert into answer (id, answer) values (default, ?)
12. Hibernate: insert into answer (id, answer) values (default, ?)

Error handling

Up until now, we threw exceptions and let failures propagate without giving this much thought. This is great and "the right thing to do". The beautiful part about exception handling is the fact that we can ignore the problem and punt it to a global area where the issue can be resolved.

The problem is that Spring Boot does not know how to display a "proper" failure to the user, as can be seen in *figure 5.8:*

Whitelabel Error Page
This application has no explicit mapping for /error, so you are seeing this as a fallback.
Sun Jan 22 13:26:04 IST 2023
There was an unexpected error (type=Not Found, status=404).
No message available

Figure 5.8: *Default Spring error page*

As the error message says, all we need to do is implement the **/error** mapping to get the default error page. The simplest thing we can do is copy the **index.html** file to **error.html** in the template directory. That means that when Spring tries to redirect to **/error**, it will reach that template and show the welcome page itself. We can edit that HTML page to show a generic error message, but we can also probably do better than that.

We can override the controller for error handling by implementing the **ErrorController** interface. Notice that this is a marker interface that contains no methods. It is there to provide the hint that we would like to handle errors ourselves. In this case, we can implement the **handleError** method as such:

1. @Controller
2. public class GenericErrorController implements ErrorController {
3. @RequestMapping("/error")
4. public String handleError(Model model, HttpServletRequest request)

```
{
5. Object status = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
6. model.addAttribute("status", status);
7. return "error";
8. }
9. }
```

This is a standard Spring MVC callback, where we set the model value. The **HttpServletRequest** is a standard Java server API that provides access to low-level networking code. With it, we can get the error information as well as everything needed in terms of the connection. When needed, we can also add that value to APIs, and Spring will inject it correctly.

Notice that we can create elaborate implementations, such as a dedicated page for 404 errors and detailed messages. In this case, we simply add the following implementation for the error page:

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Error</title>
6. </head>
7. <body>
8. <h1 th:text="${'Error Occurred: ' + status}">Error Occurred</h1>
9. <p>Get a dynamic exercise by going to
10. <a href="/question">/question</a>.</p>
11. </body>
12. </html>
```

With that, we receive the error code correctly. Notice that we can write expression code on Line 8 within the **Thymeleaf** attribute. This is a powerful tool that we can leverage extensively to better separate UIs.

Failures can be far more nuanced though. An exception in the code might represent a reasonable error message, and we might want to stop before getting into the generic error logic. In Spring Boot, you can map an error page to a specific exception by using the **@ExceptionHandler** annotation on a controller method.

The following is an example of how you can map an error page to the

NullPointerException:

```
@ControllerAdvice
public class GlobalExceptionHandler {

@ExceptionHandler(NullPointerException.class)
public ModelAndView handleNullPointerException(Exception e) {
ModelAndView modelAndView = new ModelAndView();
modelAndView.setViewName("error");
modelAndView.addObject("error", e.getMessage());
return modelAndView;
}
}
```

In this example, the handle **NullPointerException** method will handle any **NullPointerException** that is thrown in the application and return the error view with the exception message as an attribute.

In this example, we also use the **ModelAndView** class, which we did not introduce before. The **ModelAndView** class is used to represent a model and a view. A **ModelAndView** object contains both the model data (the data that needs to be displayed in the view) and the view (the template used to render the data).

A **ModelAndView** object can be created by instantiating the **ModelAndView** class and passing in the name of the view as the constructor argument. The model data can be added to the **ModelAndView** object by using the **addObject** method, which takes a key-value pair.

The following is an example of how you can create a **ModelAndView** object and add some data to it:

```
ModelAndView modelAndView = new ModelAndView("hello");
modelAndView.addObject("message", "Hello World!");
```

In this example, the **ModelAndView** object is created with the view name “hello”. The data with the key “message”, and value “Hello World!” is added to the model. You can also create a **ModelAndView** with a **ModelMap** object. This is a class that represents a simple key-value store for model attributes, as can be seen in the following code snippet:

```
ModelAndView modelAndView = new ModelAndView();
ModelMap modelMap = new ModelMap();
modelMap.addAttribute("message", "Hello World!");
```

4. modelAndView.setViewName("hello");
5. modelAndView.addAllObjects(modelMap);

Once the **ModelAndView** object is created, it can be returned from a controller method and passed to the view resolver, which will use the view name to determine the template that should be used to render the data.

Conclusion

Spring is a remarkably powerful platform that simplifies complex concepts. The same can be said for JPA, which makes the mess in SQL manageable. There are many challenges in Spring, mostly due to its depth and breadth. This book only provides a tiny glimpse into its vast landscape.

Points to remember

- Try to avoid hand-coding SQL.
- Understand the stereotypes in Spring and divide your application accordingly.
- Lombok can save us a great deal of hassle.
- The Spring Initializr provides a great start for a project.

Multiple choice questions

1. What does IoC stand for?

a. Interoperation Connector

b. Indirect Ordered Command

c. Inversion of Control

d. Instant Organized Class

e. Integrated Offering Component

2. JPA is?

a. It is more portable than SQL

b. It hides SQL

c. It is faster than SQL

d. All the above

3. Why do we use **RequiredArgsConstructor** in Beans?

a. To remove requirements?

b. To inject fields via constructor

c. To satisfy Lombok requirements

4. Which of these is a stereotype?

a. Controller

b. Component

c. RestRequest

d. RestContorller

e. Service

Answers

1. c

2. a

3. b

4. a, b, and e

---

1 https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/.

2 REpresentational State Transfer.

3 Autowired is supported but constructor injection is the recommended "best practice" in Spring.

4 https://start.spring.io/

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 6
Testing and CI

Introduction

Testing and **Continuous Integration** (**CI**) are important practices in Java development that ensure code changes are integrated smoothly without causing any unexpected errors or conflicts. Testing in Java typically involves writing automated tests to ensure that the code works as intended. This can include unit tests, which test individual units of code in isolation, as well as integration tests, which test how different parts of the system work together.

In addition to writing tests, it is important to set up CI that can automatically run those tests whenever changes are made to the codebase. This can help catch errors early in the development process before they have a chance to cause serious problems.

In this chapter, we will cover some common tools in the world of testing, such as JUnit and Mockito. We will cover GitHub Actions when discussing CI due to its wide availability and free quota. We will also discuss some high-level concepts, such as **Test Driven Development** (**TDD**).

Structure

In this chapter, we will discuss the following topics:

- Testing theory
- JUnit
- Mockito
- Test Driven Development
- Continuous Integration
- GitHub Actions

- Linting

Objectives

By the end of this chapter, you will be able to test your applications effectively and automate quality control using CI processes. You will understand the core concepts of modern testing.

Testing theory

The goal of testing is more than the mere elimination of bugs. Testing is here to assure us of compliance and continuity. With proper testing in place, we can make changes to the underlying code with greater confidence in the overall reliability. Testing is an important tool in preventing code regressions. If you work on a project where you seem to solve the same bug repeatedly, this might be a problem in the underlying testing process.

We can divide the types of tests in many ways, but here are a few important categories:

- **Unit tests**: Unit tests are the most fundamental type of tests. They are designed to test individual units of code in isolation. These tests are typically automated, and they focus on testing the smallest possible piece of functionality, such as a single method or function. Unit tests are important because they help developers catch errors early in the development process when they are easier and less expensive to fix.
- **Integration tests:** Integration tests are designed to test how different parts of a system work together. Unlike unit tests, integration tests involve testing multiple units of code at once, and they are often used to test complex systems with many moving parts. Integration tests can help catch errors that might not be caught by unit tests, such as issues with interactions between different components of the system.
- **Functional tests:** Functional tests are designed to test the functionality of a system from the perspective of the end user. These tests are often used to test the user interface or the behavior of the system under different conditions. Functional tests are important because they help ensure that the system works as expected from the user's point of view.
- **Performance tests:** Performance tests are designed to test the performance of a system under different conditions, such as under heavy load or with a large amount of data. These tests are important because they help identify performance bottlenecks and

ensure that the system can handle expected levels of usage.

- **Acceptance tests:** Acceptance tests are designed to test whether a system meets the requirements specified by the client or end user. These tests are often used to ensure that the system is functioning as intended and that it meets the needs of the user. Acceptance tests can be manual or automated, and they are often used in conjunction with other types of testing to ensure that the system is of high quality.

Overall, a combination of these types of tests is typically used to ensure that a system is of high quality and meets the needs of the user. By using a variety of tests, developers can catch errors early in the development process and ensure that the system is functioning as intended.

Unit tests are smaller, and thus, they are more performant. This means they can be executed almost at once, sometimes even in the background as you code within the IDE. This can have many advantages since an issue can be discovered and fixed early. They have an additional advantage; their errors are easy to understand. When a unit test fails, the cause is usually self-evident.

Integration tests can fail badly and run slowly. They also tend to be flakier than unit tests and often fail because of test environment issues. Flaky tests fail occasionally for no reason. When you run the test multiple times, it will fail one time out of ten. Or fail only when running after another specific test. Sometimes, this indicates a problem in the test, but sometimes it indicates an environmental problem in the software. Flaky tests are very hard to deal with as there is no reliable way to debug them.

Despite all of that, some argue that integration tests are the only tests that truly check the system. The most insidious and common bugs exist in the interconnect between systems. That area is outside of the domain of unit tests and might be missed entirely by their coverage. *Table 6.1* lists the core differences between unit and integration tests:

	Unit tests	Integration tests
Performance		
Reliability		
Coverage		
Flakiness		

Table 6.1: *Differences between unit and integration tests*

JUnit

JUnit is a popular unit testing framework for Java. It provides a set of annotations and assertions that can be used to write test cases for

individual units of code. With JUnit, developers can write automated tests that can be run quickly and easily, making it easier to catch errors early in the development process. Despite its name, JUnit is used for integration tests as well as unit tests.

JUnit is widely used in the Java development community, and it is included in many IDEs. In JUnit, tests are typically organized into test suites, which can include multiple test cases. Each test case is defined as a method with the @Test annotation, and it typically includes one or more assertions that check the behavior of the tested code.

The following code contains a simple unit test using JUnit:

```
1. package com.debugagent.testing.tests;
2. import org.junit.jupiter.api.Assertions;
3. import org.junit.jupiter.api.Test;
4.
5. public class MyTest {
6. @Test
7. public void testMyMethod() {
8. MyObject myObj = new MyObject();
9. Assertions.assertEquals(42, myObj.myMethod());
10. }
11. }
```

The test verifies that **myMethod()** returns **42** using an assertion. JUnit provides a variety of assertion methods, such as **assertEquals()** and **assertTrue()**, which can be used to check if the result of a test matches an expected value. If the value is something other than 42, an exception will be thrown, and the test will fail. We can see a successful execution is marked with a green check in *figure 6.1*:

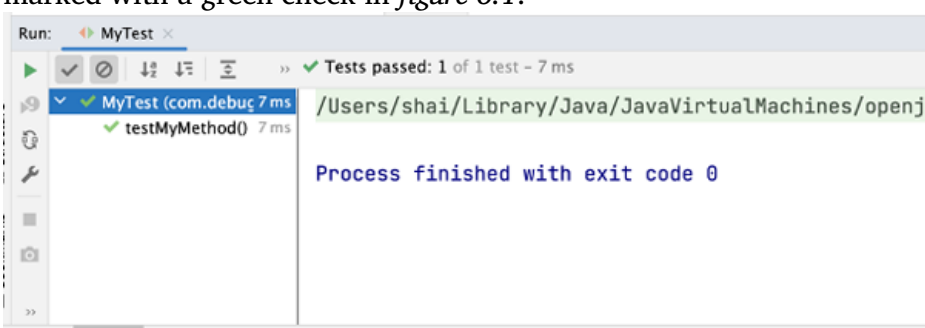

Figure 6.1: *Passing a test*

If we change the value in the test to 41, we can see the test failing in *figure*

6.2. Notice that the assertion lists the different values and can also include a custom error message.

Figure 6.2: *Passing a test*

Ideally, a test should be standalone. We should be able to run one test without running the entire suite. Alternatively, we should be able to run the tests in any arbitrary order. This is difficult to achieve in some cases. A test might leave the application in an invalid state, which can cause a problem for other tests in the suite. It is important that tests clean up after themselves. However, replicating that initialization or cleanup code through an entire suite can be painful.

JUnit provides special annotations for this exact purpose, specifically the **@Before** and **@After**, that allow you to specify methods that should be run before or after each test case in a test suite. For example, you can use **@Before** to set up resources that will be needed by all the test cases in the suite:

```
1. public class BeforeTest {
2. private MyObject myObj;
3.
4. @BeforeEach
5. public void setUp() {
6. myObj = new MyObject();
7. }
8.
9. @Test
10. public void testMyMethod() {
11. Assertions.assertEquals(42, myObj.myMethod());
12. }
```

```
13. }
```

It is important to test that methods fail in a predictable way. JUnit provides a way to test that a method throws an exception when it is called. This can be done using the **assertThrows()** method. As seen in the following example, the test verifies that an **IllegalStateException** is thrown by the code:

```
1. public class ExceptionTest {
2. @Test
3. public void testMyMethodThrowsException() {
4. MyObject myObj = new MyObject();
5. Assertions.assertThrows(IllegalStateException.class, () -> {
6. myObj.myMethodThatThrowsException();
7. });
8. }
9. }
```

Mockito

Mocking is a technique used in unit testing to create fake objects that mimic the behavior of real objects. The goal of mocking is to isolate the code being tested from its dependencies so that the tests can focus on the behavior of the code itself. The most popular Java framework mocks is Mockito. Mockito provides a simple, easy-to-use API for creating mock objects, and it is widely used in the Java development community.

With Mockito, you can create a mock object by specifying the class or interface that you want to mock and then using the **mock()** method. Once you have a mock object, you can use it to define the behavior that you want to test. Mocking is most useful in elaborate applications such as Spring Boot applications, where we can mock beans and similar functionality. If we go back to the code from *Chapter 5: It is Springtime in Java*, and we can create a test such as the following:

```
1. @SpringBootTest
2. public class HelloWebServiceTests {
3. @Mock
4. private HelloService helloService;
5.
6. @Test
7. public void testWebServicePassesValueDirectlyToService() {
8. Mockito.when(helloService.helloService("XYZ")).thenReturn("PASSED");
```

```
9. HelloWebService helloWebService = new HelloWebService(helloService);
10. Assertions.assertEquals(helloWebService.hello("XYZ"), "PASSED");
11. }
12. }
```

Notice several things about this test. We use the **@SpringBootTest** annotation so that Spring can inject into it. The **@Mock** annotation is used to inject a fake object. We can then use the **when()** method to define the behavior of the **helloService()** method.

This means that when the **helloService()** method is invoked with “XYZ” as an argument, the mock object will return “PASSED”. We then perform the operation and verify the result. This is a very simple test, but it demonstrates the power of mocking. We can use the same approach on JPA classes, or SQL calls to isolate the class and its dependencies.

The problem is that mocks hide external dependencies on the real API, and as a result, they do not test them. This limits the scope of the test to a very narrow, hard-coded set. It relies on internal implementation details such as the dependencies to implement the test, which means that the test will probably fail if the implementation changes, even though the contract does not change. Let us look at the following example:

```
1. public int countUserCities() {
2. return db.executeInt("select count("city") from users");
3. }
```

We can mock the **db** function **executeInt** since the returned result will be bad. But this will break if we change the original API call to something like the following:

```
1. public int countUserCities() {
2. return db.count("city","users");
3. }
```

This covers nothing. A far better approach is to add fake data to a temporary database[1]. We can spin up databases dynamically and “properly” check the method with a database like the one we have in production. This performs the proper check; it will fail for bugs like a typo in the query and does not rely on internal implementation.

Unfortunately, this approach is problematic. Loading a fresh database takes time. Doing it for every suite can become a problem as the scope of testing grows. That is why we separate the unit and integration tests. Performance matters.

Performance matters

Do you know what is the worst type of testing?

The ones you do not run and end up deleting.

Testing frequently is crucial; continuous testing lets us fail quickly during development. A quick failure means our mind is still fresh with the changes that triggered the failure. You do not need git bisect[2]; you can fix things almost instantly. For this to work properly, we need to run testing cycles all the time. If it takes a while to go through a testing workflow and requires some configuration (for example, docker and so on), this might collide with CPU architecture too (such as M1 Mac), and then we might have a problem.

We mock external dependencies for the unit test so that performance will improve. However, we cannot give up on the full call to the actual API because the mocking has issues. However, these can run in the CI process, and we do not need to run them all the time. Does this breed duplication: yes. It is a problem, but there is no other way. There is no workable alternative; they all end up making things worse.

Because of this, it is important to check the coverage of integration tests only. The unit test coverage is interesting but not as crucial as the integration coverage. Notice that 100% coverage is problematic, so you should not aim for that. But it is an important statistic to monitor.

What should we mock?

We used a database example before, but mocking databases might not be the best case. For Java, one would typically use a light in-memory database which works well in most cases. Fakers accept CSV formats to fill up the database and can even come up with their own fake data. This is better than mocking and lets us get close to integration test quality with unit test performance.

However, constantly connecting to a Web service dependency is impractical without mocking. In that sense, it is a good idea to mock everything that is outside of our control. In that point, we face the choice of where to mock. We can use mock servers that include coded requests and responses. This makes sense when working with an integration test. Not so much for a unit test, but we can do it if the server is stateless and local. It is preferred to rather mock the call to the API endpoints in this case, though. It will be faster than the actual API, but it could still cause problems.

Over-mocking is the process of applying mocks too liberally to too many

API calls. A developer might engage in that in order to increase the coveted coverage metric. This further strengthens our claim that coverage should not apply to unit tests as it might lead to such a situation. Behavior should not be mocked for most local resources accessed by a framework.

Test Driven Development

Test Driven Development (**TDD**) is a software development approach in which tests are written before any code is written. The development process then proceeds by running the tests, writing just enough code to pass each test, and then refactoring the code to improve its design and maintainability.

The TDD cycle typically follows the following three steps:

1. **1. Write a failing test:** The developer writes a test that describes the desired behavior of a piece of code. This test should fail because the code does not yet exist.
2. **2. Write the simplest code to pass the test:** The developer writes the simplest possible code that will make the failing test pass.
3. **3. Refactor the code:** The developer improves the code's design and maintainability by refactoring it, making it simpler and more modular while ensuring that all tests continue to pass.

By following this cycle, TDD ensures that code is written incrementally and in small, testable units. This results in code that is easier to maintain, more resilient to changes, and less prone to bugs. TDD also helps to ensure that the code meets its requirements and that any changes to the code do not break existing functionality.

TDD is often used in agile software development, where requirements can change frequently, and there is a need for rapid iteration and feedback. TDD is also used in safety-critical systems, where the cost of bugs can be very high, and in codebases, where the complexity of the code makes it difficult to test manually.

TDD is an interesting approach. It is especially useful when working with loosely typed languages. In those situations, TDD is wonderful as it fills the role of a strict compiler and linter.

There are other cases where it makes sense. When building a system that has very well-defined input and output. This is something that we have very often when creating courses and materials. When working on real-world data, this sometimes happens when we have middleware that processes data and outputs it in a predefined format.

The idea is to construct the equation with the hidden variables in the

middle. Then the coding becomes filling in the equation. It is very convenient in cases like that. Coding becomes filling in the blanks.

The problem with TDD

If we have a pre-existing system with tests, then TDD makes all the sense in the world. But testing a system that does not exist is much harder. This is especially so with languages, such as Java, that are well-structured and strict. There are some cases where it makes sense, but not as often as one would think.

The big claim for TDD is “it is design”. Tests are effectively the system design, and we then implement that design. The problem with this is that it defies debugging just like a regular design. What this leads to is an implementation that passes the test and perpetuates a bug coded by the test. TDD proponents are quick to comment that a TDD project is easier to refactor since the tests give us a guarantee that we would not have regressions. However, this applies to projects with testing performed after the fact. TDD has no advantage in this respect.

The worst problem is that TDD focuses heavily on fast unit testing. It is impractical to run slow integration tests or long run tests[3] that can run overnight on a TDD system. How do you verify scale and integration into a major system?

In an ideal world, everything will just click into place like Lego. Unfortunately, integration tests fail badly. These are the worst failures with the most difficulty in tracking bugs. It is usually better to have a failure in the unit tests, and that is why we have them. They are easy to fix. However, even with perfect coverage, they do not test the interconnect properly. We need integration tests, and they find the most terrible bugs.

TDD over-emphasizes the “nice to have” unit tests over the essential integration tests. Yes, you should have both. But we must have the integration tests.

Continuous Integration

Continuous Integration (**CI**) is a software development practice in which code changes are automatically built and tested in a frequent and consistent manner. The goal of CI is to catch and resolve integration issues as soon as possible, reducing the risk of bugs and other problems slipping into production.

CI often goes hand in hand with **Continuous Delivery** (**CD**), which aims to automate the entire software delivery process, from code integration to deployment in production. The goal of CD is to reduce the time and effort

required to deploy new releases and hotfixes, enabling teams to deliver value to customers faster and more frequently. With CD, every code change that passes the CI tests is considered to be ready for deployment, allowing teams to deploy new releases at any time with confidence. We will not discuss Continuous Delivery in this book as it is a deep rabbit hole.

Continuous Integration tools

There are many powerful CI tools. The following are some commonly used tools:

- **Jenkins:** Jenkins is one of the most popular CI tools, offering a wide range of plugins and integrations to support various programming languages and build tools. It is open-source and offers a user-friendly interface for setting up and managing build pipelines. It is written in Java and is often the go-to for many people. However, it is a pain to manage and set-up. There are some Jenkins as service solutions that also clean up its user experience, which is somewhat lacking.
- **Travis CI:** Travis CI is a cloud-based CI tool that integrates well with GitHub, making it an excellent choice for GitHub-based projects. It offers a simple setup process and provides real-time build and test results, making it easy to detect and resolve issues quickly.
- **CircleCI:** CircleCI is a cloud-based CI tool that supports a wide range of programming languages and build tools. It offers a user-friendly interface and provides real-time build and test results, making it easy to detect and resolve issues quickly.
- **GitLab CI/CD:** GitLab is a popular source code management tool that includes built-in CI/CD capabilities. With GitLab CI/CD, teams can automate their build, test, and deployment processes, making it easy to deliver code changes quickly and confidently.
- **Bitbucket Pipelines:** Bitbucket Pipelines is a cloud-based CI tool from Atlassian that integrates seamlessly with Bitbucket, their source code management tool. It offers a simple setup process and provides real-time build and test results, making it easy to detect and resolve issues quickly.

Notice we skipped GitHub Actions, which we will get to shortly. There are several factors to consider when comparing CI tools:

- **Ease of use:** Some CI tools have a simple set-up process and user-friendly interface, making it easier for developers to get started and manage their build pipelines.
- **Integration with source code management (SCM) Tools** such as

GitHub, GitLab, and Bitbucket. This makes it easier for teams to automate their build, test, and deployment processes.

- **Support for different programming languages and build tools:** Different CI tools support different programming languages and build tools, so it is important to choose a tool that is compatible with your development stack.
- **Scalability:** Some CI tools are better suited to larger organizations with complex build pipelines, while others are better suited to smaller teams with simpler needs.
- **Cost:** CI tools range in cost from free and open-source to commercial tools that can be expensive. Thus, it is important to choose a tool that fits your budget.
- **Features:** Different CI tools offer different features, such as real-time build and test results, support for parallel builds, and built-in deployment capabilities.

In general, Jenkins is known for its versatility and extensive plugin library, making it a popular choice for teams with complex build pipelines. Travis CI and CircleCI are known for their ease of use and integration with popular SCM tools, making them a good choice for small to medium-sized teams. GitLab CI/CD is a popular choice for teams using GitLab for their source code management, as it offers integrated CI/CD capabilities. Bitbucket Pipelines is a good choice for teams using Bitbucket for their source code management, as it integrates seamlessly with the platform.

Cloud versus on premise

The hosting of agents is an important factor to consider when choosing a CI solution. There are two main options for agent hosting: cloud-based and on-premise.

- **Cloud-based:** Cloud-based CI solutions, such as Travis CI, CircleCI, GitHub Actions, and Bitbucket Pipelines, host the agents on their own servers in the cloud. This means that you do not have to worry about managing the underlying infrastructure, and you can take advantage of the scalability and reliability of the cloud.
- **On-premise:** On-premise CI solutions, such as Jenkins, allow you to host the agents on your own servers. This gives you more control over the underlying infrastructure but also requires more effort to manage and maintain the servers.

When choosing a CI solution, it is important to consider your team's specific needs and requirements. For example, if you have a large and complex build pipeline, an on-premise solution such as Jenkins may be a better choice, as it gives you more control over the underlying

infrastructure. On the other hand, if you have a small team with simple needs, a cloud-based solution such as Travis CI may be a better choice, as it is easy to set up and manage.

Agent statefullness

Statefulness determines whether the agents retain their data and configurations between builds. Refer to the following:

- **Stateful agents:** Some CI solutions, such as Jenkins, allow you to run stateful agents, which means that the agents retain their data and configurations between builds. This is useful for situations where you need to persist data between builds, such as when you are using a database or running long-running tests.
- **Stateless agents:** Other CI solutions, such as Travis CI and CircleCI, use stateless agents, which means that the agents are recreated from scratch for each build. This provides a clean slate for each build, but it also means that you need to manage any persisted data and configurations externally, such as in a database or cloud storage.

There is a lively debate among CI proponents regarding the best approach. Stateless agents provide a clean and easy-to-reproduce environment.

However, they can cost as you often pay for cloud resources. They are slower to set up. But the main reason some developers prefer the stateful agents is the ability to investigate. With a stateless agent, when a CI process fails, you are usually left with no means of investigation other than the logs. With a stateful agent, we can log into the machine and try to run the process manually on the given machine. We might reproduce an issue that failed and gain insight, thanks to that.

Many such cases showed problems due to cleanup and cast a shadow over the reliability and value of our CI.

GitHub Actions

In the previous section, we skipped the discussion of GitHub Actions. This product is a relatively new contender in the field; it is not as flexible as most other CI tools mentioned before. However, it is very convenient for developers, thanks to its deep integration with GitHub and stateless agents.

To test GitHub Actions, we need a new project which in this case was generated by JHipster[4] using the configuration seen in *figure 6.3*:

```
? Which *type* of application would you like to create? Monolithic application (recommended for simple projects)

? What is the base name of your application? HelloJHipster
? Do you want to make it reactive with Spring WebFlux? No
? What is your default Java package name? com.debugagent.hellojava
? Which *type* of authentication would you like to use? JWT authentication (stateless, with a token)
? Which *type* of database would you like to use? SQL (H2, PostgreSQL, MySQL, MariaDB, Oracle, MSSQL)
? Which *production* database would you like to use? PostgreSQL
? Which *development* database would you like to use? H2 with disk-based persistence
? Which cache do you want to use? (Spring cache abstraction) Ehcache (local cache, for a single node)
? Do you want to use Hibernate 2nd level cache? Yes
? Would you like to use Maven or Gradle for building the backend? Maven
? Do you want to use the JHipster Registry to configure, monitor and scale your application? No
? Which other technologies would you like to use?
? Which *Framework* would you like to use for the client? React
? Do you want to generate the admin UI? Yes
? Would you like to use a Bootswatch theme (https://bootswatch.com/)? Default JHipster
? Would you like to enable internationalization support? No
? Please choose the native language of the application English
? Besides JUnit and Jest, which testing frameworks would you like to use?
? Would you like to install other generators from the JHipster Marketplace? No
Generating 2.048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 99,999 days
        for: CN=Java Hipster, OU=Development, O=com.debugagent.hellojava, L=, ST=, C=
```

Figure 6.3: *JHipster configuration*

Unlike the previous code, which you can find in the book's GitHub repository, in this case, we have a separate project that demonstrates the use of GitHub Actions.

Notice that you can follow this with any project; although we include maven instructions here, the concept is very simple. Once the project is created, we can open the project page on GitHub and move to the actions tab. We will see something like *figure 6.4*:

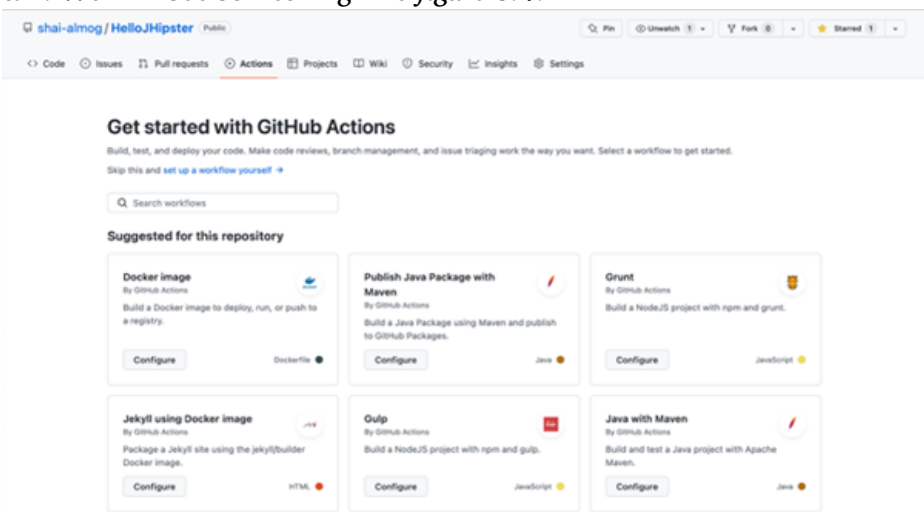

Figure 6.4: *The blank actions page*

In the bottom right corner of *figure 6.4,* we can see the Java with Maven project type. Once we pick this type, we move to the creation of a **maven.yml** file, as shown in *figure 6.5*:

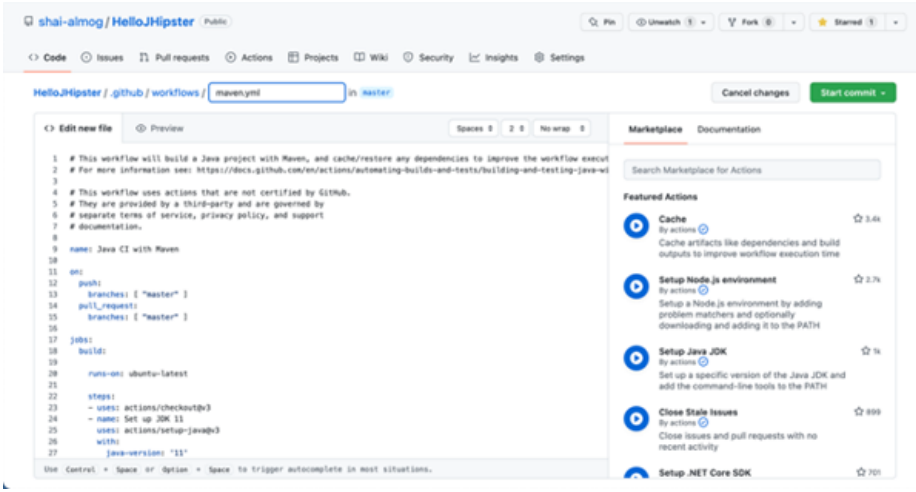

Figure 6.5: *Creating the new maven.yml*

Unfortunately, the default **maven.yml** suggested by GitHub includes a problem. This is the code seen in *figure 6.5*:

1. name: Java CI with Maven
2.
3. on:
4. push:
5. branches: ["master"]
6. pull_request:
7. branches: ["master"]
8.
9. jobs:
10. build:
11.
12. runs-on: ubuntu-latest
13.
14. steps:
15. - uses: actions/checkout@v3
16. - name: Set up JDK 11
17. uses: actions/setup-java@v3
18. with:
19. java-version: '11'
20. distribution: 'temurin'
21. cache: maven

22. - name: Build with Maven
23. run: mvn -B package --file pom.xml
24.
25. *# Optional: Uploads the full dependency graph to GitHub to improve the quality of Dependabot alerts this repository can receive*
26. - name: Update dependency graph
27. uses: advanced-security/maven-dependency-submission-action@571e99aab1055c2e71a1e2309b9691de18d6b7d6

Lines 24 and onward, update the dependency graph. However, this feature often fails. Removing them solved the problem. The rest of the code is standard YAML configuration.

Lines 5 and 7 declare that builds will run on both branches when pushing to the master branch. This means that we can run our tests on a pull request before committing. If the test fails, we will not commit. We can disallow committing with failed tests in the project settings. Once the YAML file is committed, we can create a pull request, and the system will run the build process for us. This includes running the tests since the "package" target in Maven runs tests by default and is invoked in Line 23. We can see the successful pull request[5] in *figure 6.6*:

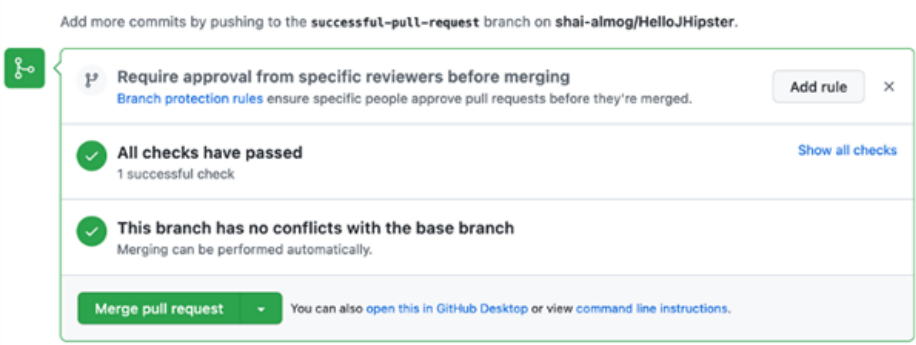
Add more commits by pushing to the successful-pull-request branch on shai-almog/HelloJHipster.
Require approval from specific reviewers before merging
Branch protection rules ensure specific people approve pull requests before they're merged.
Add rule
All checks have passed
1 successful check
Show all checks
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Figure 6.6: *Successful pull request*

To test this out, we can make a change to the code by changing the "/api" endpoint to "/myapi"[6]. This produces the failure shown in *figure 6.7*. It also triggers an error email sent to the author of the commit.

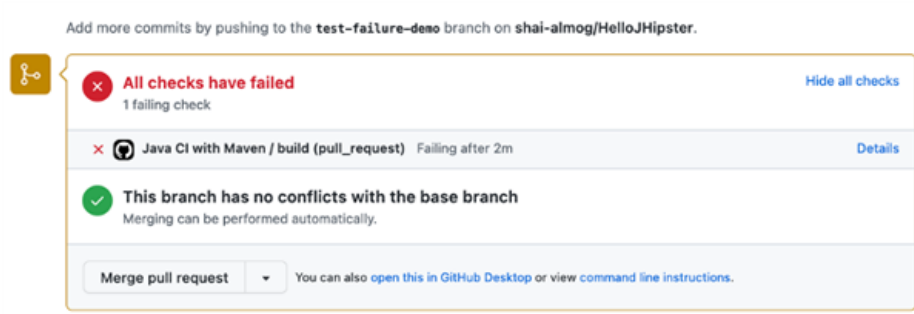
Add more commits by pushing to the test-failure-demo branch on shai-almog/HelloJHipster.
All checks have failed
1 failing check
Hide all checks
Java CI with Maven / build (pull_request) Failing after 2m
Details
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Figure 6.7: *Failed pull request*

When such a failure occurs, we can click the "**Details**" link on the right side. This takes us directly to the error message shown in *figure 6.8*:

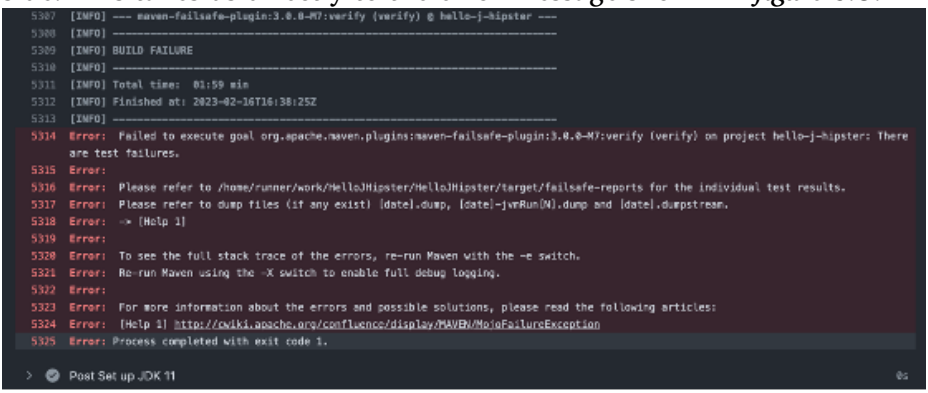
[INFO] --- maven-failsafe-plugin:3.0.0-M7:verify (verify) @ hello-j-hipster ---
[INFO] --
[INFO] BUILD FAILURE
[INFO] --
[INFO] Total time: 01:59 min
[INFO] Finished at: 2023-02-16T16:38:25Z
[INFO] --
Error: Failed to execute goal org.apache.maven.plugins:maven-failsafe-plugin:3.0.0-M7:verify (verify) on project hello-j-hipster: There are test failures.
Error:
Error: Please refer to /home/runner/work/HelloJHipster/HelloJHipster/target/failsafe-reports for the individual test results.
Error: Please refer to dump files (if any exist) [date].dump, [date]-jvmRun[N].dump and [date].dumpstream.
Error: -> [Help 1]
Error:
Error: To see the full stack trace of the errors, re-run Maven with the -e switch.
Error: Re-run Maven using the -X switch to enable full debug logging.
Error:
Error: For more information about the errors and possible solutions, please read the following articles:
Error: [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
Error: Process completed with exit code 1.
Post Set up JDK 11

Figure 6.8: *Initial error message*

Unfortunately, this is typically a useless message that does not provide help in the issue resolution. However, scrolling up will show the actual failure, which is usually conveniently highlighted for us, as seen in *figure 6.9*:

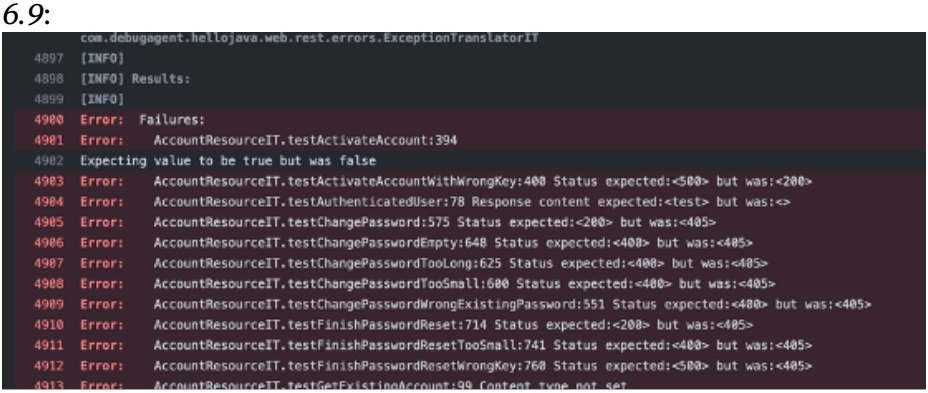
com.debugagent.hellojava.web.rest.errors.ExceptionTranslatorIT
[INFO]
[INFO] Results:
[INFO]
Error: Failures:
Error: AccountResourceIT.testActivateAccount:394
Expecting value to be true but was false
Error: AccountResourceIT.testActivateAccountWithWrongKey:400 Status expected:<500> but was:<200>
Error: AccountResourceIT.testAuthenticatedUser:78 Response content expected:<test> but was:<>
Error: AccountResourceIT.testChangePassword:575 Status expected:<200> but was:<405>
Error: AccountResourceIT.testChangePasswordEmpty:648 Status expected:<400> but was:<405>
Error: AccountResourceIT.testChangePasswordTooLong:625 Status expected:<400> but was:<405>
Error: AccountResourceIT.testChangePasswordTooSmall:600 Status expected:<400> but was:<405>
Error: AccountResourceIT.testChangePasswordWrongExistingPassword:551 Status expected:<400> but was:<405>
Error: AccountResourceIT.testFinishPasswordReset:714 Status expected:<200> but was:<405>
Error: AccountResourceIT.testFinishPasswordResetTooSmall:741 Status expected:<400> but was:<405>
Error: AccountResourceIT.testFinishPasswordResetWrongKey:760 Status expected:<500> but was:<405>
Error: AccountResourceIT.testGetExistingAccount:99 Content type not set

Figure 6.9: *The actual test failure*

Note that there are often multiple failures, so it would be prudent to scroll up further. In this error, we can see the failure was an assertion in line 394

of **AccountResourceIT**, which you can see here. Note that the line numbers do not match. In this case, Line 394 is the last line of the method, and Line 17:

```
1. @Test
2. @Transactional
3. void testActivateAccount() throws Exception {
4. final String activationKey = "some activation key";
5. User user = new User();
6. user.setLogin("activate-account");
7. user.setEmail("activate-account@example.com");
8. user.setPassword(RandomStringUtils.randomAlphanumeric(60));
9. user.setActivated(false);
10. user.setActivationKey(activationKey);
11.
12. userRepository.saveAndFlush(user);
13.
14. restAccountMockMvc.perform(get("/api/activate?
    key={activationKey}", activationKey)).andExpect(status().isOk());
15.
16. user = userRepository.findOneByLogin(user.getLogin()).orElse(null);
17. assertThat(user.isActivated()).isTrue();
18. }
```

This means the assert failed. **isActivated()** returned false and failed the test. This should help a developer narrow down the issue and understand the root cause.

Branch protection

By default, GitHub projects allow anyone to commit changes to the main (master) branch. This is problematic in most projects. We usually want to prevent commits to the master so we can control the quality of the mainline. This is especially true when working with CI, as a break in the master can stop the work of other developers.

We can minimize this risk by forcing everyone to work on branches and submit pull requests to the master. This can be taken further with code review rules that require one or more reviewers. GitHub has highly configurable rules that can be enabled in the project settings, as shown in *figure 6.10*:

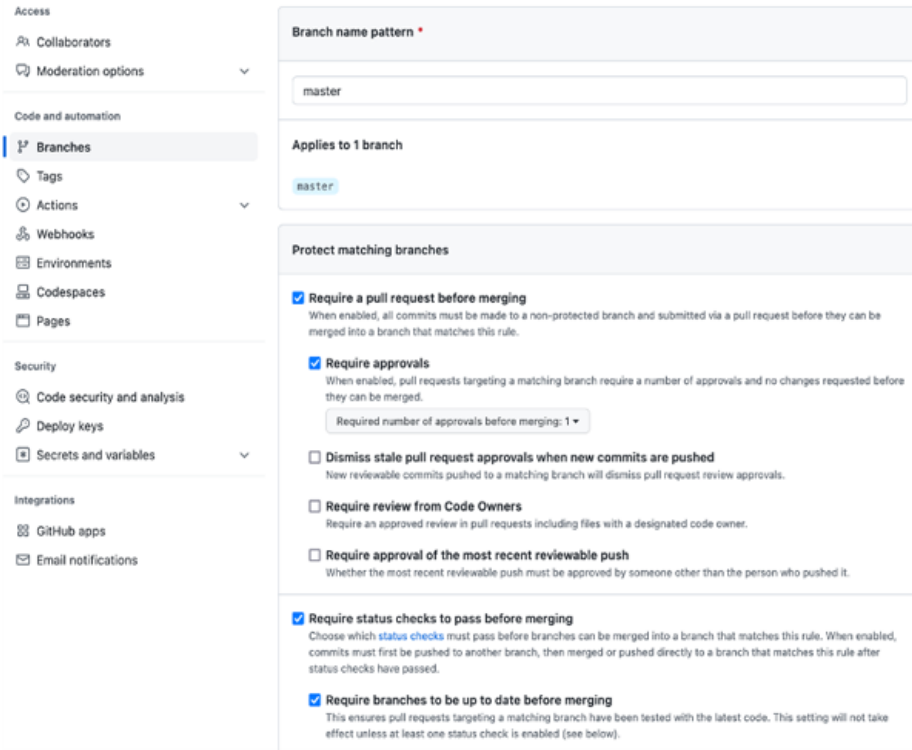
Access
Collaborators
Moderation options
Code and automation
Branches
Tags
Actions
Webhooks
Environments
Codespaces
Pages
Security
Code security and analysis
Deploy keys
Secrets and variables
Integrations
GitHub apps
Email notifications
Branch name pattern *
master
Applies to 1 branch
master
Protect matching branches
Require a pull request before merging
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.
Require approvals
When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.
Required number of approvals before merging: 1
Dismiss stale pull request approvals when new commits are pushed
New reviewable commits pushed to a matching branch will dismiss pull request review approvals.
Require review from Code Owners
Require an approved review in pull requests including files with a designated code owner.
Require approval of the most recent reviewable push
Whether the most recent reviewable push must be approved by someone other than the person who pushed it.
Require status checks to pass before merging
Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.
Require branches to be up to date before merging
This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Figure 6.10: *Branch protection rules*

Enabling branch protection on the master branch in GitHub provides several benefits, including the following:

- **Preventing accidental changes to the master branch:** By enabling branch protection on the master branch, you can prevent contributors from accidentally pushing changes to the branch. This helps to ensure that the master branch always contains stable and tested code.
- **Enforcing code reviews:** You can require that all changes to the master branch be reviewed by one or more people before they are merged. This helps to ensure that changes to the master branch are high quality and meet the standards of your team.
- **Preventing force pushes:** Enabling branch protection on the master branch can prevent contributors from force-pushing changes to the branch, which can overwrite changes made by others. This helps to ensure that changes to the master branch are made intentionally and with careful consideration.
- **Enforcing status checks:** You can require that certain criteria, such as passing tests or successful builds, are met before changes to the master branch are merged. This helps to ensure that changes to the master branch are of high quality and do not introduce new bugs or

issues.

Overall, enabling branch protection on the master branch in GitHub can help to ensure that changes to your codebase are carefully reviewed, tested, and of high quality. This can help to improve the stability and reliability of your software.

The following are a few tips for configuring branch protection in GitHub:

- **Restrict who can merge changes:** Limiting who can merge changes can help prevent unauthorized changes to your codebase. You can require that only certain people or teams are allowed to merge changes or that changes can only be merged through pull requests.
- **Enable branch deletion protection:** Enabling branch deletion protection can help prevent branches from being accidentally deleted. You can configure who is allowed to delete branches and require additional permissions or reviews before branches can be deleted.
- **Use required and optional rules in branch protection:** GitHub offers the ability to set up required and optional rules for branch protection. Required rules must be met before a pull request can be merged, whereas optional rules can be used to provide guidance or suggestions for code quality.

By carefully configuring branch protection settings, you can help ensure that changes to your codebase are high-quality, reviewed, and tested before they are merged. This can help improve the stability and reliability of your software.

Linting

SonarCloud is a cloud-based version of SonarQube that allows developers to continuously inspect and analyze their code to find and fix issues related to code quality, security, and maintainability. It supports various programming languages such as Java, C#, JavaScript, Python, and more. SonarCloud integrates with popular development tools such as GitHub, GitLab, Bitbucket, Azure DevOps, and more. Developers can use SonarCloud to get real-time feedback on the quality of their code and improve the overall code quality.

On the other hand, SonarQube is an open-source platform that provides static code analysis tools for software developers. It provides a dashboard that shows a summary of the code quality and helps developers to identify and fix issues related to code quality, security, and maintainability. SonarQube supports a wide range of programming languages, such as

Java, C#, JavaScript, Python, and more. It also provides various plugins and extensions to extend its functionalities. SonarQube can be integrated with popular development tools such as Jenkins, Maven, Gradle, and more.

Both SonarCloud and SonarQube provide similar functionalities, but SonarCloud is a cloud-based service and requires a subscription, whereas SonarQube is an open-source platform that can be installed on-premise or on a cloud server. For simplicity's sake, we will use SonarCloud, but SonarQube should work just fine.

To get started, we go to sonarcloud.io and signup, ideally with our GitHub account. We are then presented with an option to add a repository for monitoring by Sonar Cloud, as shown in *figure 6.11*:

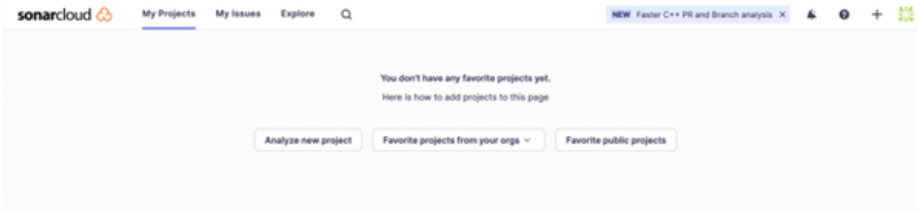
sonarcloud
My Projects
My Issues
Explore
NEW Faster C++ PR and Branch analysis
You don't have any favorite projects yet.
Here is how to add projects to this page
Analyze new project
Favorite projects from your orgs
Favorite public projects

Figure 6.11: *Initial Sonar Cloud page after signup*

When we select the **Analyze new project** option, we need to authorize access to our GitHub repository. The next step is selecting the projects we wish to add to Sonar Cloud, as shown in *figure 6.12*:

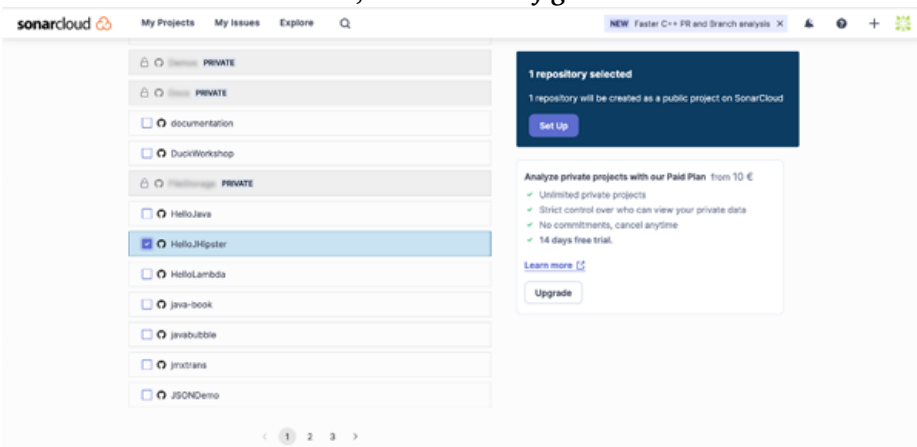

Figure 6.12: *Enabling sonar Cloud to the HelloJHipster project*

Once we select and proceed to the setup process, we need to pick the analysis method. Since we use GitHub Actions, we need to pick that option in the following stage, as seen in *figure 6.13*:

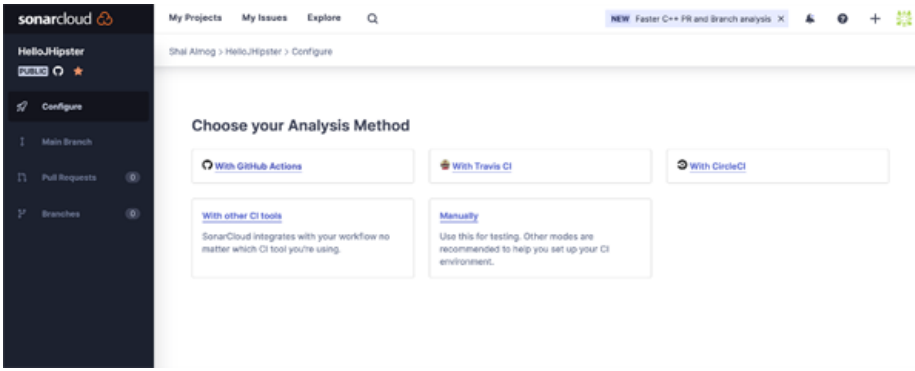

Figure 6.13: *Picking the analysis method*

Once this is set, we enter the final stage within the Sonar Cloud wizard, as shown in *figure 6.14*. We receive a token that we can copy (entry 2 that is blurred in the image); we will use that shortly. Notice that there are also default instructions to use with Maven that appear once you click the button labeled "**Maven**".

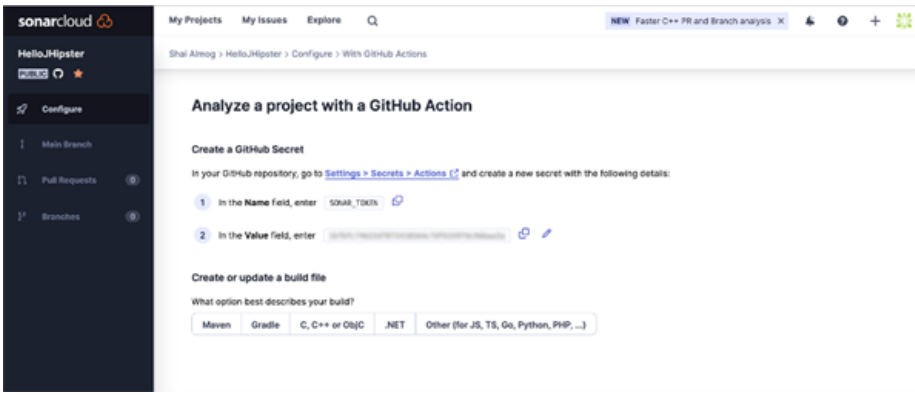
sonarcloud
My Projects
My Issues
Explore
NEW Faster C++ PR and Branch analysis
HelloJHipster
PUBLIC
Shai Almog > HelloJHipster > Configure > With GitHub Actions
Configure
Main Branch
Pull Requests
Branches
Analyze a project with a GitHub Action
Create a GitHub Secret
In your GitHub repository, go to Settings > Secrets > Actions and create a new secret with the following details:
1 In the Name field, enter SONAR_TOKEN
2 In the Value field, enter
Create or update a build file
What option best describes your build?
Maven
Gradle
C, C++ or ObjC
.NET
Other (for JS, TS, Go, Python, PHP, ...)

Figure 6.14: *Picking the analysis method*

Going back to project in GitHub, we can move to the **Project settings** tab (not to be confused with the **Account settings** in the top menu). Here we select "**Secrets and variables**" as shown in *figure 6.15*:

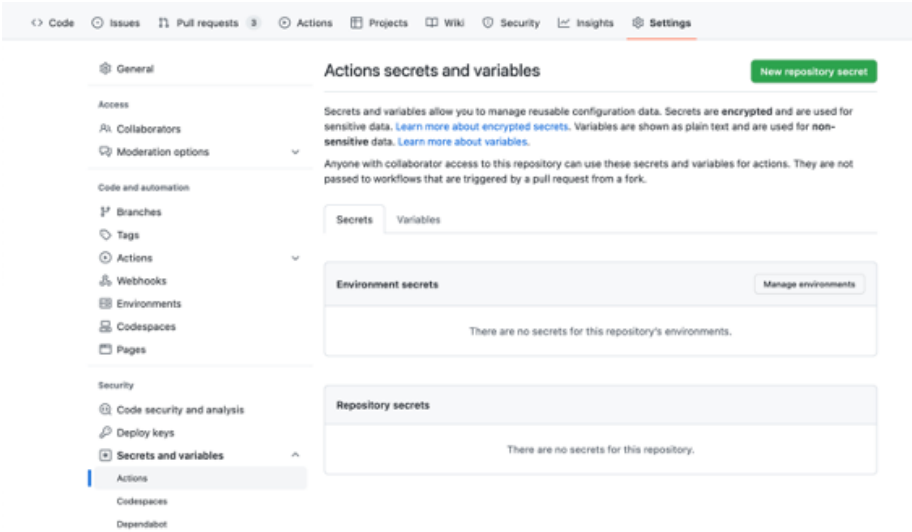

Figure 6.15: *Action secrets and variables*

In this section, we can add a new repository secret, specifically the **SONAR_TOKEN** key and value we copied from the SonarCloud, as you can see in *figure 6.16*:

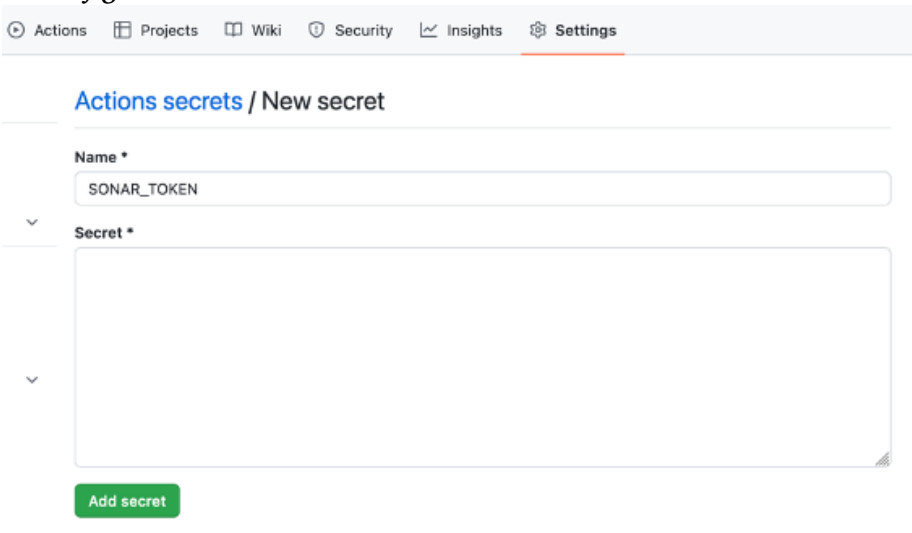

Figure 6.16: *Picking the analysis method*

GitHub Repository Secrets are a feature that allows developers to securely store sensitive information associated with a GitHub repository, such as API keys, tokens, and passwords, which are required to authenticate and authorize access to various third-party services or platforms used by the repository.

The concept behind GitHub Repository Secrets is to provide a secure and

convenient way to manage and share confidential information, without having to expose the information publicly in code or configuration files. By using secrets, developers can keep sensitive information separate from the codebase and protect it from being exposed or compromised in case of a security breach or unauthorized access.

GitHub Repository Secrets are stored securely and encrypted and can only be accessed by authorized users who have been granted access to the repository. Secrets can be used in workflows, actions, and other scripts associated with the repository, and can be passed as environment variables to the code, so that it can access and use the secrets in a secure and reliable way.

Overall, GitHub Repository Secrets provide a simple and effective way for developers to manage and protect confidential information associated with a repository, thus helping to ensure the security and integrity of the project and the data it processes.

We now need to integrate this into the project. First, we need to add these two lines to the **pom.xml** file. Notice that you need to update the organization name to match your own. These should go into the <**properties**> section in the XML:

1. <sonar.organization>shai-almog</sonar.organization>
2. <sonar.host.url>https://sonarcloud.io</sonar.host.url>

Notice that the JHipster project we created, already has SonarQube support, which should be removed from the pom file before this code will work[7].

After this, we can replace the "**Build with Maven**" portion of the **maven.yml** file with the following version:

```
1. - name: Build with Maven
2.     env:
3.       GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get
         PR information, if any
4.       SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
5.     run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-
       plugin:sonar -Dsonar.projectKey=shai-almog_HelloJHipster
       package
```

Once we do that, SonarCloud will provide reports for every pull request merged into the system, as shown in *figure 6.17*:

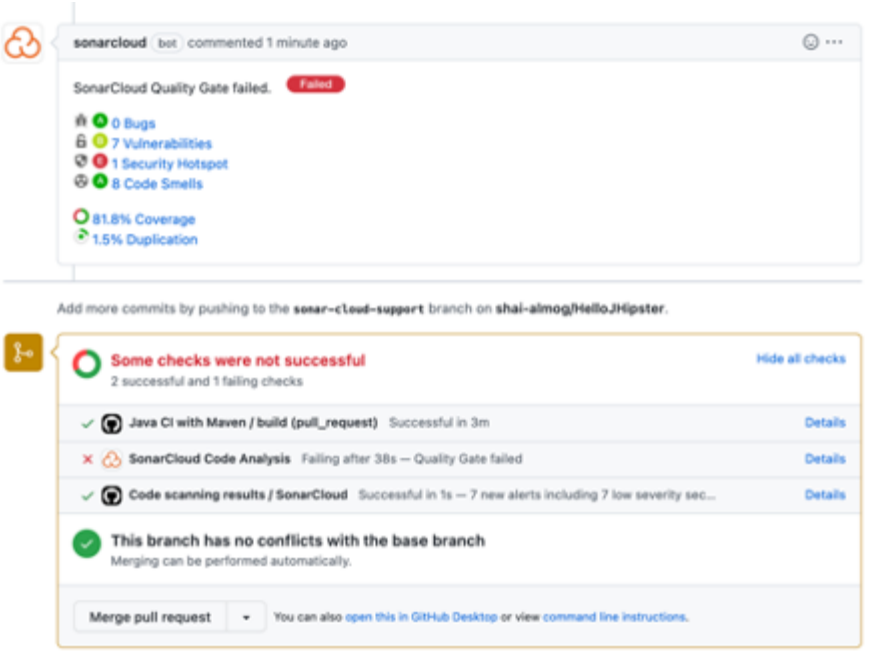
sonarcloud bot commented 1 minute ago
SonarCloud Quality Gate failed.
Failed
0 Bugs
7 Vulnerabilities
1 Security Hotspot
8 Code Smells
81.8% Coverage
1.5% Duplication
Add more commits by pushing to the sonar-cloud-support branch on shai-almog/HelloJHipster.
Some checks were not successful
2 successful and 1 failing checks
Hide all checks
Java CI with Maven / build (pull_request) Successful in 3m
Details
SonarCloud Code Analysis Failing after 38s — Quality Gate failed
Details
Code scanning results / SonarCloud Successful in 1s — 7 new alerts including 7 low severity sec...
Details
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Figure 6.17: *SonarCloud report for a pull request*

We can see a report that includes a list of bugs, vulnerabilities, smells, and security issues. Clicking every one of those issues leads us to something like *figure 6.18*:

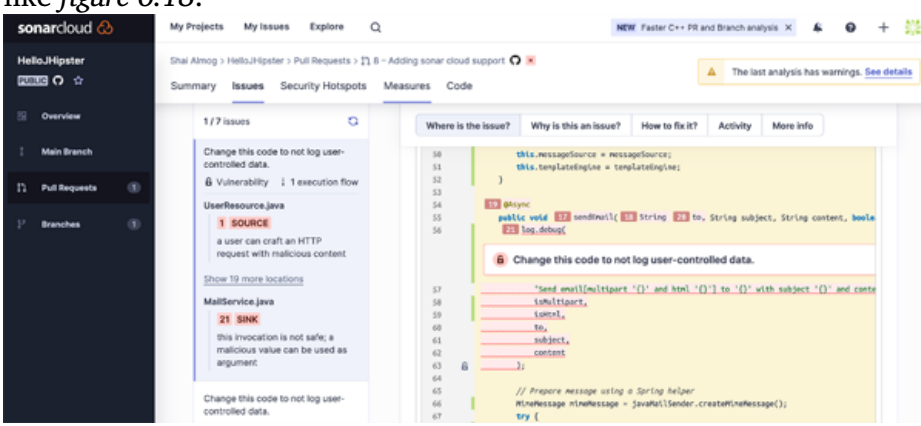

Figure 6.18: *SonarCloud issue report*

Notice that we have tabs that explain exactly why the issue is a problem, how to fix it, and more. This is a remarkably powerful tool that serves as one of the most valuable code reviewers on the team.

Two additional interesting elements we saw in *figure 6.17* are the coverage and duplication reports. SonarCloud expects that tests will have 80% code coverage (trigger 80% of the code in a pull request); this is high and can

be configured in the settings. It also points out duplicate code which might indicate a violation of the **Don't Repeat Yourself** (**DRY**) principle. We can see the coverage report in *figure 6.19*. We can then use that report to target the weak spots in our tests.

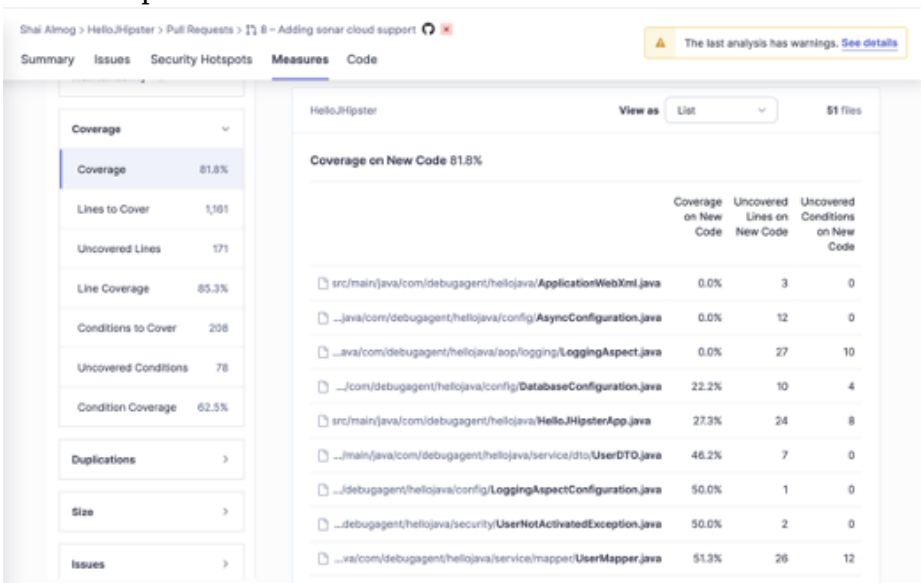

Figure 6.19: *Coverage report*

Once we merge in a few commits, we can use the dashboard in SonarCloud to get an overview of the project's health.

Conclusion

Testing and CI are crucial for Java developers to ensure the reliability and quality of our code, speed up the development process, and reduce the risk of bugs and issues in the final product. GitHub Actions and SonarCloud bring code quality to a whole new level.

Points to remember

- Writing automated tests is essential to ensure that your Java application works correctly and continues to work as changes are made.
- Unit tests are focused on testing individual units of code, such as methods or classes, in isolation from the rest of the application.
- Integration tests verify that different components of the application work correctly together.
- By using mocking in your testing, you can reduce the complexity of your tests, make them more focused, and increase their reliability. It also enables you to catch issues early in the development process,

which helps reduce the time and effort required to fix them later on.

- CI tools such as Jenkins, Travis CI, and CircleCI automate the process of building and testing code changes, allowing developers to identify and fix issues more quickly.
- By incorporating testing and CI into the development process, developers can catch issues earlier, reduce the risk of bugs and defects in the production environment, and ultimately deliver higher-quality software.

Multiple choice questions

1. JUnit is used for?

a. Integration tests

b. Unit tests

c. Performance tests

d. All the above

e. Integration and Unit tests

2. What is the problem with unit tests?

a. Incomplete coverage

b. Slow performance

c. Difficult setup

d. Unreliable

e. Hard to read results

3. Why enable branch protection? (Choose all that apply)

a. Stop hackers from changing our code

b. Prevent accidental changes to the master branch

c. Enforcing code reviews

d. It is a GitHub Actions requirement

4. What is a repository secret?

a. The hidden directory where the GitHub Actions YAML file is stored

b. The API key we pass between GitHub and vendors (for example, SonarCloud)

c. An arbitrary value we can expose to CI without including it in the code

d. The clickthrough license we agreed to when signing up

Answers

1. e

2. a

3. b, c

4. c

---

1 This is easy thanks to projects such as Testcontainers https://www.testcontainers.org/

2 https://talktotheduck.dev/understand-the-root-cause-of-regressions-with-git-bisect

3 Longrun tests are integration tests that are meant to stress out the system on heavy load and can sometimes run all night to verify integrity and scale.

4 JHipster is an open-source code generator that creates sample enterprise projects typically in Spring Boot. You can read about it here: https://www.jhipster.tech/

5 https://github.com/shai-almog/HelloJHipster/pull/4

6 https://github.com/shai-almog/HelloJHipster/pull/7

7 You can see the details of that in this pull request https://github.com/shai-almog/HelloJHipster/pull/8

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 7
Docker, Kubernetes, and Spring Native

Introduction

The original thought behind this chapter was around cloud-native development. However, that acronym is somewhat vague. In the old days, we would take the resulting jar files and place them on an application server connected to the internet. However, those days are long gone; containers and orchestration have completely changed the way applications are deployed in production.

These deployment methods triggered a seismic shift down the chain. Applications need to be smaller and load faster. This gave rise to tools such as GraalVM, and Spring Native, which is built on top of it. In this chapter, we will learn how to build applications for modern cloud deployment and what exactly that means.

Structure

In this chapter, we will discuss the following topics:

- Containers
- Orchestration
- Infrastructure as Code (IaC)
- Spring Native

Objectives

In this chapter, we will learn how to build and deploy modern applications using virtualization, orchestration, and Spring Native.

Containers

In the past, deploying complex applications in production required the physical setup of a machine or using telnet to connect to it. This process was complicated and error-prone, particularly when setting up on multiple sites. Virtualization solved this problem by allowing developers to create virtual machines, replicating the $\times 86$ OS environment. This made it easier to set up and deploy applications, and also allowed cloud server providers to sell space on their servers. However, virtualization has the following two drawbacks:

1. Significant overhead
2. Lack of standardization across vendors

Containers were developed to solve these issues. They use kernel capabilities to isolate running code, thus creating a separate environment that "feels" like a separate machine. Containers can run on top of virtual machines with minimal overhead, and a standard container format has made it easy to move container images around.

In *figure 7.1*, we show the long road from hardware to virtualization, containers, and orchestration. During that journey, we traded convenience, portability, raw performance, and scalability.

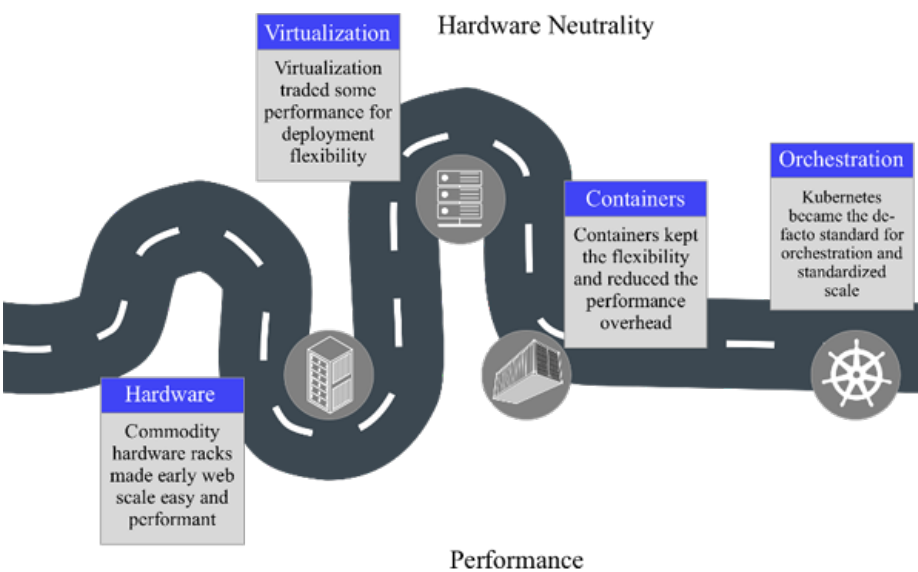
Virtualization
Virtualization traded some performance for deployment flexibility
Hardware Neutrality
Containers
Containers kept the flexibility and reduced the performance overhead
Orchestration
Kubernetes became the de-facto standard for orchestration and standardized scale
Hardware
Commodity hardware racks made early web scale easy and performant
Performance

Figure 7.1: *The road to Orchestration*

Containers have enabled massive scale and reduced costs by allowing for more containers on the same hardware, particularly with distroless or bare containers that do not include the full operating system.

However, containers do not cover everything, including dynamic fail-over, updates, and scaling. Solutions above the container runtime are required to ensure service uptime and provision additional resources dynamically.

Docker

Docker is a containerization platform that allows you to create, deploy, and run applications in containers. It is the product that brought the concept of containers to the forefront. Docker provides a way to package an application and its dependencies into a single container, thus making it easy to deploy and run the application on any machine or environment that supports Docker.

Container technology is not new and was not invented by Docker. The revolution Docker offered was in the simplicity of container creation. Docker lets anyone define a simple image using a standardized process. Prior to this, only Linux administrators had the skills required to set up a container. Thanks to Docker, this is a trivial task, and we can spin up a database, application, and so on, with very little work. Docker uses a client-server architecture, where the Docker client communicates with the Docker daemon, which is responsible for building, running, and managing Docker containers.

The first step we need to follow is installing a Docker desktop, which we can perform by following the instructions on their website: **https://docs.docker.com/desktop/** matching the specific OS platform.

Once Docker is installed, we can verify that the installation worked by issuing the following command:

1. docker version

This should display information about the Docker client and daemon versions. If it displays an error or fails, check the following:

- Review the installation steps and make sure you covered all the bases.
- Did you run the docker process? Make sure you run Docker Desktop or on Linux "**sudo systemctl start docker**".
- If the Docker daemon is running, but you are still unable to use the docker command, it is possible that the Docker daemon is not accessible. You can check if the Docker daemon is accessible by running the following command: **docker info**.
- If you are running Docker as a non-root user, make sure that your user account is added to the Docker group. This will give you permission to run Docker commands without using **sudo**. You can

add your user account to the docker group by running the following command: **sudo usermod -aG docker <your-username>**. Replace **<your-username>** with your actual username.

Next, we need to build a Docker image. Docker images are built from a Dockerfile, which is a text file (without an extension) that contains instructions for building the image. We create a new directory called **myapp** and create a file called Dockerfile with the following contents:

```
1. FROM openjdk:11
2.
3. COPY . /app
4.
5. WORKDIR /app
6.
7. RUN javac HelloWorld.java
8.
9. CMD ["java", "HelloWorld"]
```

This Dockerfile specifies a base image of OpenJDK 11, copies the application files into the **/app** directory, sets the working directory to **/app**, compiles the **HelloWorld.java** file, and sets the command to run the compiled **HelloWorld** class.

Create a new file called **HelloWorld.java** with the following contents:

```
1. public class HelloWorld {
2. public static void main(String[] args) {
3. System.out.println("Hello, World!");
4. }
5. }
```

This file defines a simple Java application that prints a “Hello, World!” message to the console. Now that you have the files ready, run the following command to build the Docker image:

```
1. docker build -t myapp .
```

This command builds a Docker image with the name **myapp**, using the Dockerfile in the current directory. We now created a Docker image. The image is the description of how a container instance looks. We can use one image to create multiple server instances and scale applications easily.

Now that we have a Docker image, it is time to run a containerized instance of your application. We run the following command to start a new

container from the myapp image:

1. docker run myapp

This command starts a new container from the myapp image. You should see the “Hello, World!” message displayed in your console. We can use the following:

1. docker ps

This shows the list of containers that are running in docker, and then we can stop this container using the following command:

1. docker stop <container-id>

Replace <**container-id**> with the ID of the container listed in the **docker ps** command. Notice that this is a very simplified scenario. In this example, we even perform the compilation within the container which includes the code. As we progress along the chapter, we will show a better way.

What we did here was launch a separate server that ran in isolation on our machine. It is running in a “clean” environment that has a fresh installation of Linux. That means that we have no subtle dependencies on our current machine, which will fail when we deploy in production. We can take the docker image and replicate the container to any machine anywhere and get similar results. That is the first big power of Docker.

The second one is this: notice this line from the docker file: **FROM openjdk:11**. This line hides a great deal of functionality; JDK 11 is already installed on the image, and we can rely on it. We can use such images to build on top of work from other developers without installing every package for every case. But it gets even better than that.

If our app requires a database, we can define a dependency on the database image and build on top of that. We can build a dependency between the images and can separate them to keep proper isolation.

Orchestration

As we discussed before, containers do not cover problems such as dynamic fail-over, updates, and scaling. We need solutions above the container runtime, in order to ensure service uptime and provision additional resources dynamically.

Container orchestration refers to the management and automation of containerized applications in a distributed computing environment. It involves deploying, scaling, managing, and monitoring containers and container clusters, in order to ensure that they run efficiently, reliably, and securely.

Container orchestration is necessary because modern applications are typically complex, distributed, and composed of multiple microservices running in containers. Manually managing these containers is time-consuming and error-prone, and can lead to problems such as inconsistent deployments, lack of scalability, and difficulty in updating or rolling back applications.

Container orchestration tools automate many of these tasks, allowing developers and DevOps teams to focus on higher-level tasks such as designing and building applications. Container orchestration platforms provide features such as load balancing, auto-scaling, self-healing, service discovery, centralized management, and monitoring. This makes it easier to deploy and manage containerized applications in a distributed environment.

Kubernetes (k8s)

Kubernetes is also known as k8s due to the eight characters between the letters K and s in Kubernetes. It is an open-source container orchestration platform that was originally developed by Google and is now maintained by the **Cloud Native Computing Foundation** (**CNCF**). Kubernetes automates the deployment, scaling, and management of containerized applications, making it easier to manage and scale complex applications in a containerized environment. It is the leading Orchestration solution in the market at present.

Kubernetes was first introduced by Google in 2014 and was released as an open-source project the following year. Since then, Kubernetes has become one of the most popular container orchestration platforms, with a large and active community of developers and contributors.

Kubernetes was originally based on Google's internal container orchestration platform, called Borg, which has been in use for over a decade. Kubernetes was designed to bring the benefits of Borg to the broader community and to provide a common platform for container orchestration that could be used across different cloud providers and on-premises data centers.

Today, Kubernetes is widely used in production environments, powering some of the largest and most complex applications and services on the internet. Its popularity and success can be attributed to its ability to simplify the management of containerized applications, improve efficiency and reliability, and provide a consistent platform for developers and operations teams.

Most of us do not need Kubernetes[1]. It was built to solve problems at

Google's scale of deployments, problems of scale that most of us will never see. It is often the case that Kubernetes is chosen as a part of **Resume Driven Design** (**RDD**). Despite that, there is a case for Kubernetes that goes beyond scale. The secondary value is standardization into deployable units. This lets us use a set of standardized tools and deployment strategies when monitoring and managing the cloud.

To get started with Kubernetes, we need to first take a long way around and then show a simpler approach. It is often more conductive to the learning experience if we understand the underlying process and not just the simplified version. We will walk through the steps of setting up a Kubernetes cluster, deploying an application, and scaling it.

Typically, we use cloud-based Kubernetes hosts when provisioning. Using those would make this book overly specific to a single provider. Instead, we will use the Kubernetes support built-in to Docker Desktop for ease of use. After installing Docker Desktop, open the application and go to **Preferences | Kubernetes**. Check the box to enable Kubernetes and click **Apply & Restart**. *Figure 7.2* shows Kubernetes enabled in Docker desktop on the Mac:

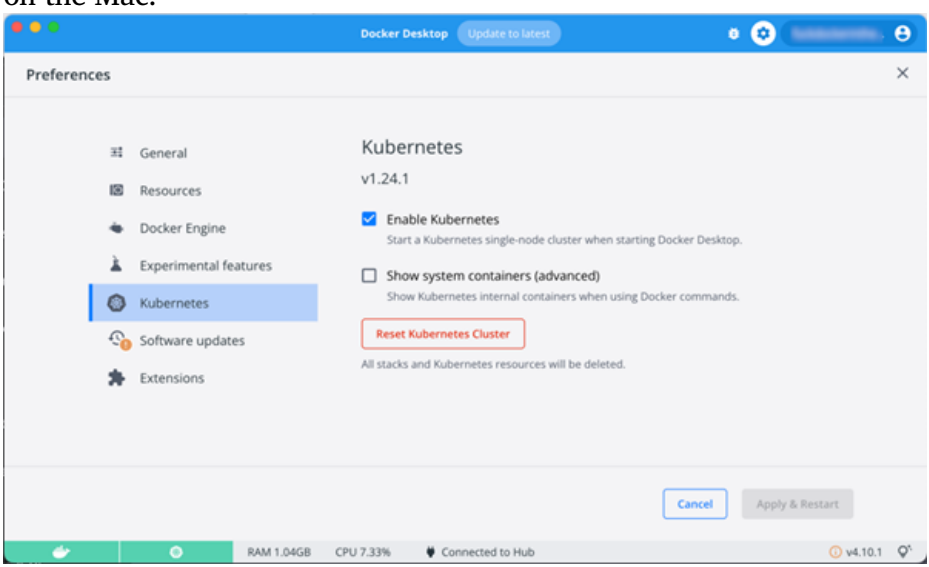

Figure 7.2: *Docker desktop settings with Kubernetes enabled*

Enabling this will start a Kubernetes cluster in the Docker desktop, which you can use to deploy and manage applications.

Next, we need to create a Kubernetes deployment. In Kubernetes, a deployment is a way to manage a set of identical pods. Pods are the smallest deployable units in Kubernetes and can contain one or more containers. To create a deployment, we should define a deployment manifest. The following is an example deployment manifest for a Java

application:

1. apiVersion: apps/v1
2. kind: Deployment
3. metadata:
4. name: my-java-app
5. spec:
6. selector:
7. matchLabels:
8. app: my-java-app
9. replicas: 3
10. template:
11. metadata:
12. labels:
13. app: my-java-app
14. spec:
15. containers:
16. - name: my-java-app
17. image: openjdk:11-jre
18. ports:
19. - containerPort: 8080
20. env:
21. - name: SPRING_PROFILES_ACTIVE
22. value: production

This might be somewhat overwhelming but if you look at the YAML file, the configuration is very clear. Most of the attributes in the YAML file consist of metadata: descriptions, classifications, labels, and so on. The end of the YAML looks like a standard docker file containing the details of our application.

The only attribute that is special is in Line 9: **replicas**. The **replicas** attribute is used to specify the desired number of copies for a given deployment. A deployment is a higher-level abstraction that manages a set of identical pods. The **replicas** attribute tells Kubernetes how many replicas of the pod it should maintain at any given time.

In this example, the **spec.replicas** attribute is set to 3, which means that Kubernetes will ensure that three replicas of the pod are specified in the **template.spec.containers** section are running at all times.

When you update the number of replicas for a deployment, Kubernetes will automatically scale the number of replicas up or down, in order to match the desired state. For example, if you update the **replicas** attribute from 3 to 5, Kubernetes will create two new replicas of the pod to bring the total to 5. If you then update the replicas attribute to 2, Kubernetes will terminate three of the running replicas to bring the total down to 2.

The replicas attribute is a powerful tool for managing the availability and scalability of your applications in Kubernetes. By specifying the desired number of replicas, you can ensure that your application is always available, even if some replicas fail or are taken offline for maintenance. You can also use the replicas attribute to scale your application up or down based on changing demand.

Continuing with our example we can now save the YAML file to our computer as **my-java-app.yaml**. This deployment manifest specifies a deployment named **my-java-app** with three replicas. Each replica runs a container with the **openjdk:11-jre** image and exposes port 8080. It also sets the environment variable **SPRING_PROFILES_ACTIVE** to production. This is a common trick to differentiate between production and debug environments; that way we can connect to a different database server and segregate local, staging, and production environments.

We can deploy locally with the following command:

1. kubectl apply -f my-java-app.yaml

This will create the deployment and start the specified number of replicas. Next, we need to expose the deployment as a service. A Kubernetes service is an abstraction that defines a logical set of pods and a policy by which to access them. Services enable network access to pods and provide load balancing.

When a client sends a request to the service, the load balancer selects one of the available replicas of the application to handle the request. If a replica becomes unavailable, the load balancer automatically routes traffic to the remaining replicas, ensuring that the application remains available even if some replicas fail.

Kubernetes load balancing provides a simple and powerful way to distribute traffic across multiple replicas of your application, improving availability and scalability. By using services to provide load balancing, you can ensure that clients can always access your application, regardless of the underlying infrastructure.

To expose the deployment as a service, we need to define a service manifest. The following is an example service manifest for the **my-java-**

app deployment:

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4. name: my-java-app
5. spec:
6. selector:
7. app: my-java-app
8. ports:
9. - name: http
10. protocol: TCP
11. port: 80
12. targetPort: 8080
13. type: LoadBalancer
```

We can save this YAML file to our computer as **my-java-app-service.yaml**. This service manifest specifies a service named **my-java-app**, which selects pods with that label. It also exposes port 80 and forwards traffic to port 8080 on the pods. Finally, it sets the service type to LoadBalancer. We can create this service using the following command:

```
1. kubectl apply -f my-java-app-service.yaml
```

This will create the service and make the **my-java-app** deployment accessible from outside the cluster.

The easy way—Skaffold

Skaffold is a popular open-source command-line tool for streamlining the development and deployment of containerized applications. With Skaffold, developers can automate the build, test, and deployment process for their applications in a consistent and repeatable way, making it easier to deploy to Kubernetes clusters.

Skaffold provides a declarative configuration that allows developers to define the build context, container image, deployment target, and other settings for their applications. This configuration can be stored in a YAML file and version controlled along with the application code.

Some of the key features of Skaffold include the following:

- **Fast, iterative development:** Skaffold can watch for changes in your code and automatically rebuild and redeploy your application to Kubernetes, making it easy to iterate quickly during

development.

- **Built-in support for common tools and frameworks:** Skaffold integrates with popular tools and frameworks such as Docker, Kubernetes, Helm, and Google Cloud Build, thus making it easy to incorporate these tools into your development workflow.
- **Consistent and repeatable builds:** Skaffold provides a consistent and repeatable way to build and deploy your application, ensuring that your deployments are always predictable and reliable.
- **Support for multiple deployment targets:** Skaffold supports deploying your application to a local Kubernetes cluster, a remote Kubernetes cluster, or a cloud provider such as Google Cloud Platform or Amazon Web Services.

While we can use Skaffold in the command line, this is a vast subject. Instead, we will use the Google Cloud Code plugin for IntelliJ/IDEA, which can be installed in the settings UI, as shown in *figure 7.3*:

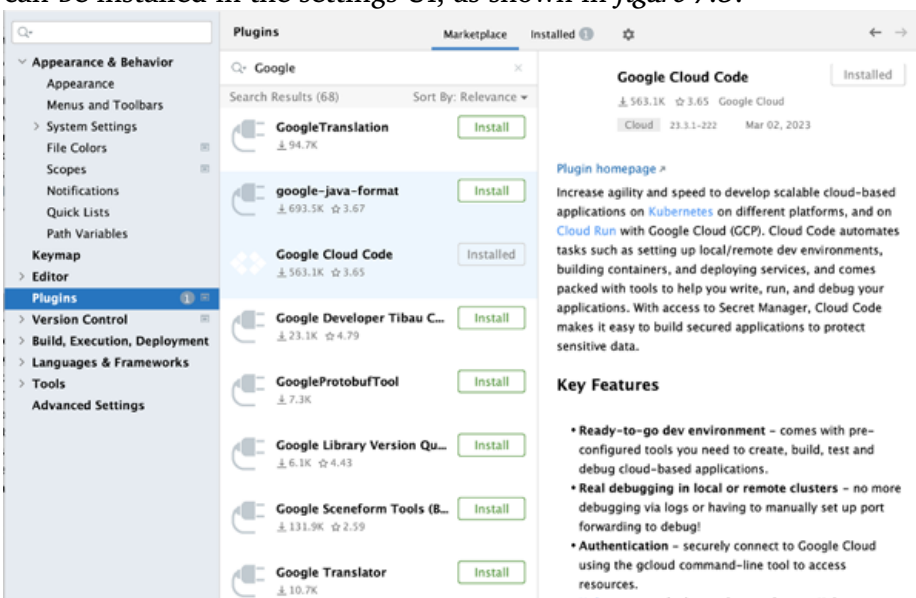
Plugins
Marketplace
Installed
Appearance & Behavior
Appearance
Menus and Toolbars
System Settings
File Colors
Scopes
Notifications
Quick Lists
Path Variables
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Advanced Settings
Google
Search Results (68)
Sort By: Relevance
GoogleTranslation
94.7K
Install
google-java-format
693.5K 3.67
Google Cloud Code
563.1K 3.65
Installed
Google Developer Tibau C...
23.1K 4.79
GoogleProtobufTool
7.3K
Google Library Version Qu...
6.1K 4.43
Google Sceneform Tools (B...
131.9K 2.59
Google Translator
10.7K
Google Cloud Code
563.1K 3.65 Google Cloud
Cloud
23.3.1-222
Mar 02, 2023
Plugin homepage
Increase agility and speed to develop scalable cloud-based applications on Kubernetes on different platforms, and on Cloud Run with Google Cloud (GCP). Cloud Code automates tasks such as setting up local/remote dev environments, building containers, and deploying services, and comes packed with tools to help you write, run, and debug your applications. With access to Secret Manager, Cloud Code makes it easy to build secured applications to protect sensitive data.
Key Features
Ready-to-go dev environment - comes with pre-configured tools you need to create, build, test and debug cloud-based applications.
Real debugging in local or remote clusters - no more debugging via logs or having to manually set up port forwarding to debug!
Authentication - securely connect to Google Cloud using the gcloud command-line tool to access resources.

Figure 7.3: *Google Cloud Code Plugin in IntelliJ/IDEA*

Once installed we can create a new project and select the Java “hello world” sample, as shown in *figure 7.4*:

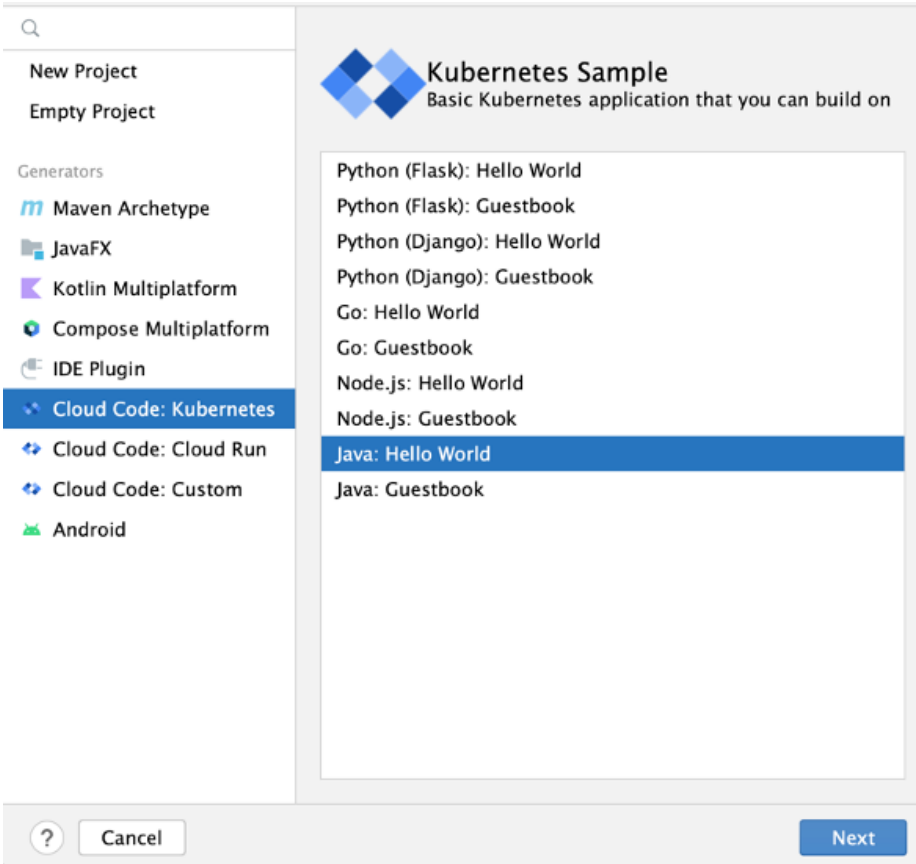

Figure 7.4: *Kubernetes sample from the Google Cloud Code Plugin in IntelliJ/IDEA*

Once this is set, you are faced with a new Spring Boot project within IntelliJ/IDEA. This is a standard Spring Boot project that can be configured with the **scaffold.yaml** and the matching Dockerfile:

1. *1. # Use maven to compile the java application.*
2. FROM maven:3-jdk-11-slim AS build-env
3.
4. *4. # Set the working directory to /app*
5. WORKDIR /app
6.
7. *7. # copy the pom.xml file to download dependencies*
8. COPY pom.xml ./
9.
10. *10. # download dependencies as specified in pom.xml*
11. *11. # building dependency layer early will speed up compile time when pom is unchanged*

```
RUN mvn verify --fail-never

14. # Copy the rest of the working directory contents into the container
COPY . ./

17. # Compile the application.
RUN mvn -Dmaven.test.skip=true package

20. # Build runtime image.
FROM openjdk:11.0.16-jre-slim

23. # Copy the compiled files over.
COPY --from=build-env /app/target/ /app/

26. # Starts java app with debugging server at port 5005.
CMD ["java", "-jar", "/app/hello-world-1.0.0.jar"]
```

The Docker file builds and packages the application, and then runs the jar in the last line of the project. Once we run this project, it will package everything and set up a local environment to test. It will even connect to a remote debugger seamlessly through port forwarding if you press the debug button. *Figure 7.5* shows the "hello world" sample that we see when the application loads successfully:

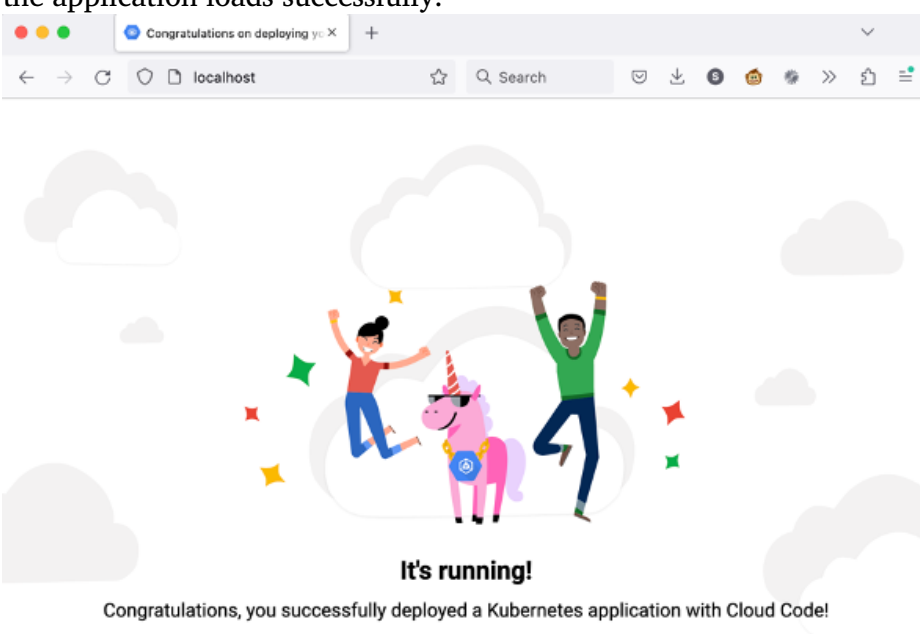

Figure 7.5: *Skaffold application running on localhost Port 80*

This can fail for the following two main reasons:

- **Missing Kubernetes runtime:** Make sure Docker Desktop is running and Kubernetes is enabled.
- **Maven missing from the system path:** This might require a reboot.

Alternatively, you can edit the run configuration to the project and modify the path element to include the maven/bin directory, as seen in *figures 7.6* and *7.7*:

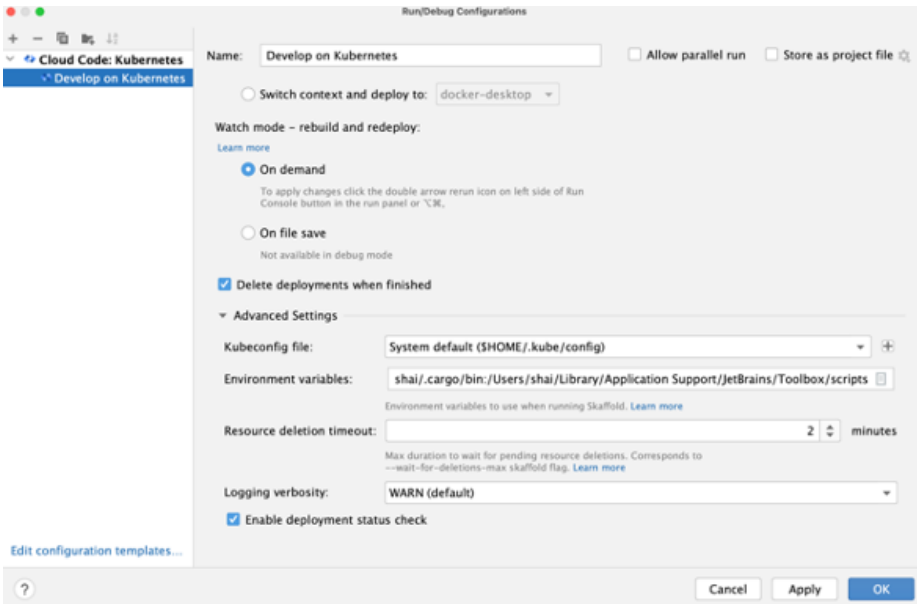

Figure 7.6: *Run configuration settings for Skaffold project*

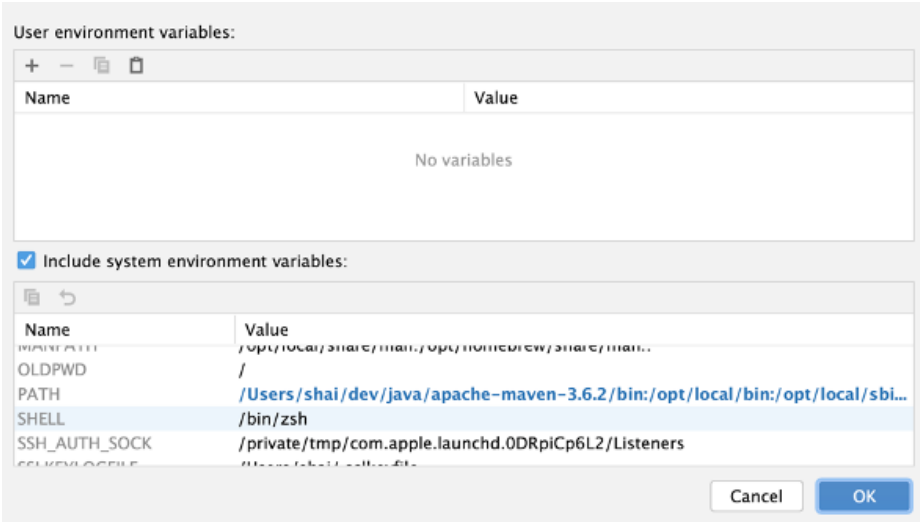

Figure 7.7: *Environment variables settings, the path is highlighted and can be edited*

With this out of the way, we can deploy Spring applications to Kubernetes with ease.

Infrastructure as Code (IaC)

Up until now, we looked at Kubernetes and Docker from the angle of a software developer and not as infrastructure. These are tools that lie at the seams between our work and our production. They are typically the domain of DevOps engineers.

When we started using cloud services, we used to configure elements in clunky Web user interfaces. Sometimes we use service-specific command lines or API tools. However, that was very problematic.

Infrastructure as Code (**IaC**) is an approach to managing infrastructure where the entire infrastructure is described in code, just like any other software. This means that infrastructure can be version-controlled, tested, and deployed using software development practices, which brings numerous benefits such as increased agility, reliability, and scalability.

We can implement IaC, with many tools such as Terraform, CloudFormation, Ansible, and many more. We will focus on Terraform since this is a large enough subject. Terraform is a popular open-source tool for provisioning and managing infrastructure.

First, we need to install Terraform. We can download the latest version of Terraform from the official website at **https://www.terraform.io/downloads.html** and follow the installation instructions.

We should then initialize Terraform by navigating to the project directory

and running the command "**terraform init**". This will download the required provider plugins and initialize our Terraform environment.

The provider is a plugin that Terraform uses to interact with a specific cloud provider. For example, if you want to provision infrastructure on AWS, you will need to configure the AWS provider. We can create a file called "**provider.tf**" in your project directory and add the following configuration:

```
1. provider "aws" {
2. region = "us-east-1"
3. }
```

This configures the AWS provider for the **us-east-1** region. You can change the region to any other region that your AWS account has access to.

Next, we need to create a resource configuration. A resource is a piece of infrastructure that Terraform manages, such as an EC2 instance, a VPC, or a security group. Create a file called "**resource.tf**" in your project directory and add the following configuration:

```
1. resource "aws_instance" "example" {
2. ami = "ami-0c94855ba95c71c99"
3. instance_type = "t2.micro"
4. tags = {
5. Name = "example-instance"
6. }
7. }
```

This configures an EC2 instance with the **Amazon Machine Image** (**AMI**) "**ami-0c94855ba95c71c99**" and the instance type "**t2.micro**". It also adds a tag to the instance with the key "**Name**" and the value "**example-instance**."

To make your Terraform code more modular and reusable, you can define variables that can be passed in at runtime. Create a file called **variables.tf** in your project directory and add the following configuration:

```
1. variable "aws_access_key" {}
2. variable "aws_secret_key" {}
```

We can pass keys and secrets here that we do not commit to the repository. That way, we can keep private information secure and easily updatable.

In the project directory, we can now run the command **terraform apply**. This will prompt you to review the changes that Terraform will make to your infrastructure, and if you approve them, Terraform will provision the

resources.

This means that we can push a major infrastructure change and go through infrastructure code reviews. This reduces the chances of a mistake by a “fat finger” on the Web UI. It also lets us reuse deployment code. We can use the same Terraform script to deploy on staging, dev, and production. That way our environments will be almost identical. This is where the variable options are very handy.

Spring Native

Spring Native is a project that aims to bring Spring applications to cloud-native platforms in a more efficient and streamlined way. The motivations for Spring Native stem from the increasing adoption of cloud-native architectures, which require lightweight, scalable, and portable applications that can be easily deployed and managed in a containerized environment.

One of the key motivations for Spring Native is to improve the performance and memory footprint of Spring applications in cloud-native environments. Traditional Java applications often suffer from high startup times and memory usage due to the heavy weight of the **Java Virtual Machine** (**JVM**), and the extensive use of reflection and classpath scanning in the Spring framework. Spring Native addresses these issues by leveraging native-image technology, which compiles Java code into a native executable that can be run directly on the operating system without the need for a JVM.

Another motivation for Spring Native is to simplify the deployment and management of Spring applications in containerized environments. With Spring Native, developers can create self-contained executable files that contain all the necessary dependencies and can be easily deployed in a container image without any additional setup or configuration. This makes it easier to package, distribute, and deploy Spring applications in a consistent and reproducible way, which is essential in cloud-native environments.

Furthermore, Spring Native aims to provide a seamless and familiar experience for Spring developers who are used to the Spring programming model and ecosystem. Spring Native integrates with the existing Spring tooling and frameworks, such as Spring Boot, Spring Cloud, and Spring Data, and provides additional features and optimizations that are specifically designed for cloud-native platforms.

While Spring Native brings several benefits to Spring applications in cloud-native environments, there are also some challenges and limitations that

developers should be aware of. The following are some of the main problems with Spring Native:

- **Limited compatibility:** Spring Native relies on the GraalVM native-image compiler to generate native executables, which has some limitations in terms of language features, runtime behavior, and third-party library support. Some Spring features, such as dynamic proxies, reflection, and AOP, may not be fully supported or may require additional configuration. Additionally, some third-party libraries or frameworks may not work well with native-image and may require special handling or customization.
- **Increased build complexity:** Generating native executables with GraalVM requires additional configuration and dependencies that may increase the complexity of the build process. For example, developers need to specify the native-image properties, include the necessary reflection and resource files, and optimize the memory usage and startup time of the application. This may require additional tooling, knowledge, and testing to ensure that the native executable works correctly and efficiently.
- **Longer build times:** Building native executables with GraalVM can take significantly longer than building traditional Java applications due to the complexity of the compilation process and the large number of dependencies and classes that need to be analyzed. This can slow down the development and testing cycle and may require additional hardware resources or parallelization to speed up the build.
- **Increased maintenance burden:** Native executables generated with GraalVM may require more maintenance and monitoring than traditional Java applications due to their different runtime behavior and memory usage. Developers need to be aware of the potential memory leaks, performance issues, and security vulnerabilities that may arise from running native executables and may need to use specialized profiling and monitoring tools to detect and diagnose problems. This is a bigger problem since the monitoring and observability capabilities of the JVM are top-notch and the GraalVM executables are not in the same league at the time of this writing.
- **Limited platform support:** Spring Native compiles to a native OS target. As such, it will be more challenging to target a different OS or CPU architecture and will require separate CI/CD pipelines. A good example is the rise of ARM CPUs. Your local machine might be an ARM machine, but you might be deploying it to an Intel production server (or vice versa). This makes server migrations harder.

Getting started with Spring Native

Most Spring projects start in the Spring Initializr[2] project, where we must add the Spring Native support option, as shown in *figure 7.8*:

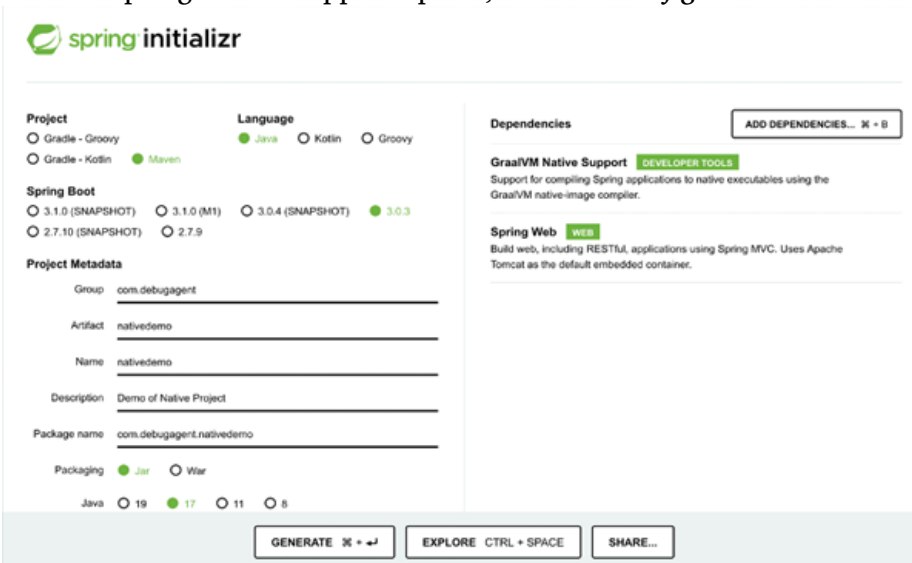

Figure 7.8: *Spring Initializr with GraalVM Native support dependency*

In the project, we need to add the GraalVM native support for the native image. Compiling a Spring Native application in the IDE is a painful experience. The build process is flaky due to several different reasons:

- The VM used for the build must be GraalVM and there are several different places in which it should be set. It should also be first on the system path which might be inconvenient.
- The **JAVA_HOME** environment variable must point at GraalVM.
- We need to use the "native" profile.

This might not be a dealbreaker as we would often use GraalVM in the CI process. The local build might be too slow for day-to-day development. On the command line, the process of building the native image simply requires the following command:

1. mvn -pnative native:build

This produces a native executable in the target directory that can be run similarly to the native jar. To give you a sense of the difference between Spring Native and the JVM version of Spring, check out *figure 7.9*, which highlights the output of running with JDK 19, and *figure 7.10*, which shows the same application running with GraalVM. Note that these numbers were generated on a Mac Book M1 but are consistent across

platforms.

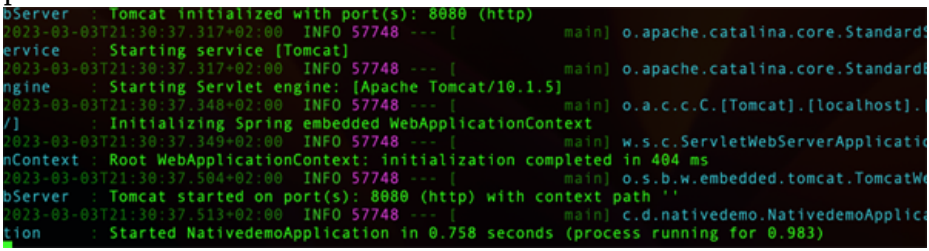
bServer : Tomcat initialized with port(s): 8080 (http)
2023-03-03T21:30:37.317+02:00 INFO 57748 --- [main] o.apache.catalina.core.Standard
ervice : Starting service [Tomcat]
2023-03-03T21:30:37.317+02:00 INFO 57748 --- [main] o.apache.catalina.core.Standard
ngine : Starting Servlet engine: [Apache Tomcat/10.1.5]
2023-03-03T21:30:37.348+02:00 INFO 57748 --- [main] o.a.c.c.C.[Tomcat].[localhost].
/] : Initializing Spring embedded WebApplicationContext
2023-03-03T21:30:37.349+02:00 INFO 57748 --- [main] w.s.c.ServletWebServerApplicati
nContext : Root WebApplicationContext: initialization completed in 404 ms
2023-03-03T21:30:37.504+02:00 INFO 57748 --- [main] o.s.b.w.embedded.tomcat.TomcatW
bServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-03-03T21:30:37.513+02:00 INFO 57748 --- [main] c.d.nativedemo.NativedemoApplic
tion : Started NativedemoApplication in 0.758 seconds (process running for 0.983)

Figure 7.9: *Spring Native demo running with OpenJDK 19*

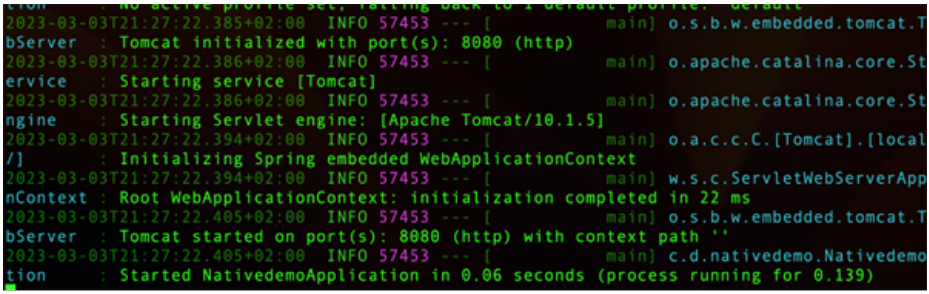
2023-03-03T21:27:22.385+02:00 INFO 57453 --- [main] o.s.b.w.embedded.tomcat.T
bServer : Tomcat initialized with port(s): 8080 (http)
2023-03-03T21:27:22.386+02:00 INFO 57453 --- [main] o.apache.catalina.core.St
ervice : Starting service [Tomcat]
2023-03-03T21:27:22.386+02:00 INFO 57453 --- [main] o.apache.catalina.core.St
ngine : Starting Servlet engine: [Apache Tomcat/10.1.5]
2023-03-03T21:27:22.394+02:00 INFO 57453 --- [main] o.a.c.c.C.[Tomcat].[local
/] : Initializing Spring embedded WebApplicationContext
2023-03-03T21:27:22.394+02:00 INFO 57453 --- [main] w.s.c.ServletWebServerApp
nContext : Root WebApplicationContext: initialization completed in 22 ms
2023-03-03T21:27:22.405+02:00 INFO 57453 --- [main] o.s.b.w.embedded.tomcat.T
bServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-03-03T21:27:22.405+02:00 INFO 57453 --- [main] c.d.nativedemo.Nativedemo
tion : Started NativedemoApplication in 0.06 seconds (process running for 0.139)

Figure 7.10: *Spring Native demo compiled to Native by GraalVM*

The same holds true for memory results in *figure 7.11*, where we can see the overhead of the application running in JDK 19, and *figure 7.12* shows the same application compiled using GraalVM:

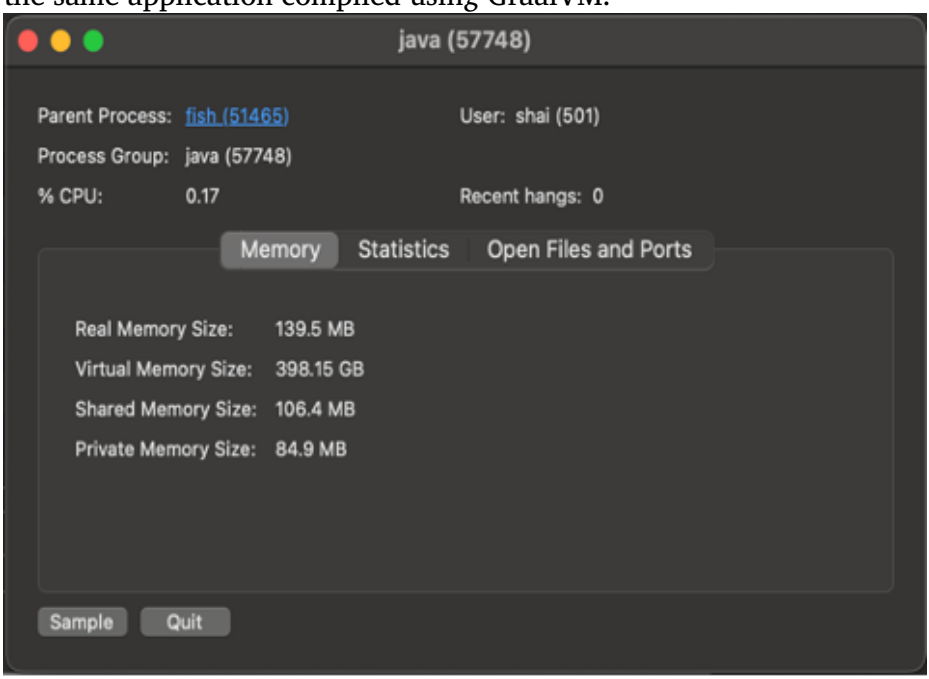

Figure 7.11: *Memory view of the Spring Native demo running with OpenJDK 19*

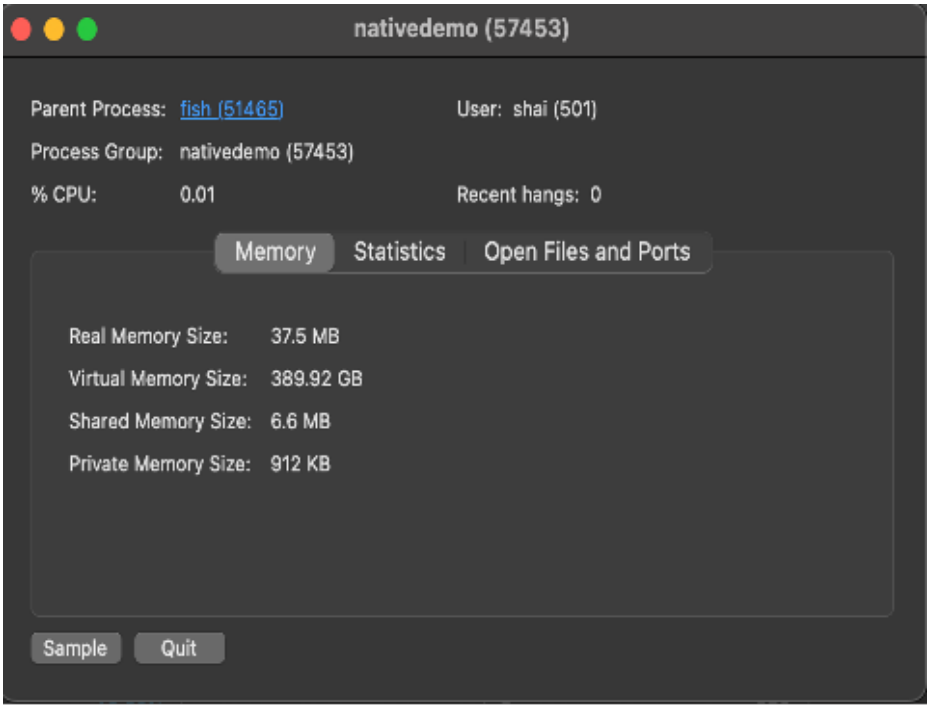

Figure 7.12: *Memory view of the Spring Native demo compiled to Native by GraalVM*

These results might be hard to read; for convenience, they are listed in the following table:

Table 7.1: *JVM versus GraalVM*

These numbers are not representative. The simpler the application, the greater the advantage GraalVM has. In real-world scenarios, do not expect a 12,000% difference in performance. However, both startup time and memory are much better with GraalVM.

Cloud Native

You might be wondering why startup time and RAM are important to us, and what they might have to do with Kubernetes or Docker.

The answer relates to the direction of cloud-native. Cloud-native is an approach to building and running applications that take advantage of the scalability, flexibility, and resilience of cloud computing. At its core, cloud-

native emphasizes the use of containerization, microservices architecture, and continuous delivery to enable faster development, deployment, and scaling of modern applications.

We will discuss microservices in greater detail in the next chapter. For now, an inaccurate simplification would be that, instead of creating one big application, we create many small applications.

For example, let us assume we want to push an update to a system with many different replicated microservices. We will need to replace all the old versions at some point. Every such update means computing time, and this can add up quickly. If our system is experiencing heavy load, we might need to spin up additional instances to handle the load and the speed of loading them might be a determining factor. If a system loads quickly, we can set Kubernetes to have a low initial count of replicas because we know it can spin up additional resources quickly. However, for a slow system, we will need to keep more replicas on standby and might idle cloud resources.

Part of that replication process includes physically copying the image data to a new physical machine. The larger the image, the slower the process. DevOps engineers have reduced images to their barebones, literally. Production images often have no tools within them, and a developer has no access to such images. All in the name of size. By reducing the size of the build, Spring Native makes the startup time of a replica even faster.

Reducing memory is of tremendous importance as well. There are obvious reasons; if we save on memory, we can use more memory for features such as caching. However, in the case of cloud computing, there is an even more powerful and reachable goal. Replications might occupy the same machine, and by reducing the total RAM usage, we can have more replicas in a single physical machine and scale faster as a result.

Alternatives to Spring Native

There are several alternatives to Spring Native, and they can be divided into two camps:

1. Alternatives to Spring Boot that rely on GraalVM.
    - Solutions that offer GraalVM advantages on top of the JVM

GraalVM has advantages in startup time and memory footprint, but these come at a price: build time increase, observability limits, restrictions on dynamic capabilities, and so on. There are two big projects geared to offer a smaller memory footprint and faster startup time within Java.

The first is project Lilliput[3] whose goal is to shrink the JVM RAM overhead by reducing object headers. Prior to Lilliput, every object in Java

had a 128-byte overhead. This adds up, as you can see as follows:

```
1. // This will take 4 bytes
2. int example = 0;
3.
4. // This will take at least 20 bytes (16 byte header + 4 byte int)
5. Integer obj = 0;
6.
7. // This will take 32 bytes (16 byte array header + 4 bytes length + 4
   bytes * 3 ints)
8. int[] parray = { 1, 2, 3 };
9.
10. // This will take 80 bytes (16 byte array header + 4 bytes length + 20
    bytes * 3 ints)
11. Integer[] oarray = { 1, 2, 3 };
```

The problem becomes far more severe with scale and multiple collections and the overhead listed here is on the conservative minimalist size. At this time, Lilliput aims to reduce the overhead of each object to 64-bit (8 bytes) but might even be able to reduce the overhead to 32-bit (4 bytes). If it can reach the first goal, the Integer in the previous example would take 12 bytes instead of 20 and the object array in that example would take 48 bytes instead of 80.

If Lilliput can reach its more ambitus goal of 4-byte overhead per object, an Integer will take up 8 bytes instead of 20. In such a situation, the object array will take up 32 bytes; this is less than half the size in both cases!

This will go even further with the changes coming from Project Valhalla, which will reduce the overhead of smaller objects. By converting objects to stack entries, their overhead might be eliminated completely.

The solution for startup time is radically different. The codename for this solution is **Coordinated Restore at Checkpoint** (**CRaC**)[4] and it is based on the **Checkpoint/Restore in Userspace** (**CRIU**) feature available in Linux[5].

Startup time is spent loading and initializing the application state. With CRaC, we can grab a snapshot of the application state and save it to disk. When we want to run the application, the system can load that snapshot and the app would be ready to go immediately. No loading code needs to run. In samples published by Azul (the specification leader), a Spring Boot application that took 3,898 ms to load, took 38 ms using CRaC[6]. That is an astounding improvement in startup time.

As you might expect, this improvement comes with a cost. The JVM needs to grab a “restore point” where it can suspend the JVM and grab a snapshot. The main problem is that connections or open files will not work. A network connection that is open to a server will reference a connection that is long gone and irrelevant. The application will need to ensure all connections and files are closed (including database connections) before saving the restore point.

Luckily, most frameworks keep track of all connections and can handle those aspects of CRaC almost seamlessly for you. Spring already has some initial support for CRaC that can be incorporated if the standard becomes official.

Conclusion

Cloud-native includes many aspects, some of them in direct conflict with one another, for example, portability and observability versus size and performance. Some of these tools are the purview of DevOps engineers, which is great. However, understanding the full stack all the way down to the hardware can be tremendously beneficial even for a high-level language developer.

Points to remember

- Containers let us abstract the system on which we will run and hide the differences in a portable way without sacrificing performance.
- Container orchestration refers to the management and automation of containerized applications in a distributed computing environment.
- Most of us do not need Kubernetes. It was built to solve problems at Google’s scale of deployments, problems of scale that most of us will never see.
- Typically, we use cloud-based Kubernetes hosts when provisioning.
- When a client sends a request to the service, the load balancer selects one of the available replicas of the application to handle the request. If a replica becomes unavailable, the load balancer automatically routes traffic to the remaining replicas, ensuring that the application remains available even if some replicas fail.
- **Infrastructure as code** (**IaC**) is an approach to managing infrastructure where the entire infrastructure is described in code, just like any other software. This means that infrastructure can be version-controlled, tested, and deployed using software development practices, which brings numerous benefits such as increased agility, reliability, and scalability.

- Spring Native relies on the GraalVM native-image compiler to generate native executables, which has some limitations in terms of language features, runtime behavior, and third-party library support.
- Cloud-native is an approach to building and running applications that takes advantage of the scalability, flexibility, and resilience of cloud computing.

Multiple choice questions

1. What is the relation between containers and virtualization (choose all that apply)?

a. Technological successors

b. Containers are enhanced virtualization

c. Containers can run on the virtual server

d. Containers have lower overhead

e. Virtualization has a lower overhead

2. Orchestration offers:

a. Replication

b. Fault tolerance

c. Scaling

d. Deployment

e. All of the above

3. How is Spring Native different from Spring Boot?

a. It is native

b. It runs in Docker

c. It targets Kubernetes

d. It is compiled with GraalVM

4. Project Lilliput aims to:

a. Move objects from the heap to the stack

b. Reduce the size of object headers

c. Reduce object startup time via snapshotting

d. Align with Project Gulliver

Answers

1. a, c, and d

2. e

3. d

4. b

---

1 See: doyouneedkubernetes.com for more details.

2 https://start.spring.io/

3 https://wiki.openjdk.org/display/lilliput

4 https://openjdk.org/projects/crac/

5 https://criu.org/Main_Page

6 https://foojay.io/today/introducing-the-openjdk-coordinated-restore-at-checkpoint-project/

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 8
Microservices

Introduction

The rise of Kubernetes has made the Microservices approach far more commonplace. The cost of spinning up and managing multiple smaller servers became manageable, and as a result, drove a massive move in that direction. Microservices bring a great deal to the table; they are easier to build and often easier to scale.

The complexity is still present in Microservices and is often greater than before. The complexity did not go away, rather, it shifted into the networking stack.

One of the biggest problems is the loose definition of Microservices, which leads to the small Monoliths anti-pattern. A bad practice that highlights the worst of both worlds.

Structure

In this chapter, we will discuss the following topics:

- Microservices versus small Monoliths
  - Service mesh
  - Authentication and authorization
  - Eventual consistency
  - Messaging
- Monolith first
- Modular Monolith

Objectives

In this chapter, we will learn about the various system architecture choices

we can make. We will also learn how we can identify a good implementation of a Monolith or Microservice architecture.

Microservices versus small Monoliths

First, we need to understand what differentiates Microservice architecture from Monolithic architecture. There is a common misconception that breaking up a Monolith simply means splitting a large server. As the definition of Microservices is so abstract, this might be technically correct, but probably not a good idea overall.

A good place to start is an authoritative definition:

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

— James Lewis and Martin Fowler (2014) [1]

Another authority on Microservices is AWS itself, whose definition is similar but has some interesting subtext:

"*Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams.*

Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features."

— AWS Website [2]

These definitions would fit a Monolith that we split into multiple smaller instances. However, if we read between the lines, we can see that this is an oversimplification of the quote. In a standard Monolithic architecture, our dependencies and impact can be carried all over the place. Splitting up a Monolith based on business logic will provide the benefit of smaller projects, but it will still leave a lot of the problems of the Monolith. In fact, it will make them worse.

Let us extract the keywords from the definition and then we can work our way back to understanding what constitutes a good Microservice:

- **Architectural and** organizational **approach:** This is probably the most important aspect of Microservices. It means that we organize the code and team in a particular way is a significant aspect. In such

a situation we can divide our team based on tiers. For example, a team that implements the database layer and so on. In Microservices, the division is based on smaller **isolated business verticals**.

- **Independent Services:** Another key aspect is the independence of the service. The word "Independent" carries with it a lot of weight in that definition. It means the **implementation should be hidden**; it is just a detail. It means that if the service fails or crashes, things should keep working as expected, we need to **isolate** the failure.
- **Self-contained:** Continues the path started with independent services. That means that if our team builds a Microservice and wants to deploy said Microservice, we can just do that. There is no need to wait to deploy the entire application; we can focus on our small domain. This implies several things. First, **deployment should be automated with CD**. It also means that **deployment should be independent** of other Microservices. Otherwise, the cost of deploying hundreds of Microservices will not scale. Finally, it means that the system **should be decentralized**. Without that, we would deploy a new version, and no one would know. A single point of failure could bring down everything without this.

There are several principles here but there is one that is missing from the definition and is very important: observability. With a monolithic application, we can open the logs or inspect the server details. When we have hundreds of types of servers, this becomes untenable. We need tooling that can help with that problem, hence, observability. As we will discuss observability in *Chapter 10, Monitoring and Observability*, we will skip the details of that aspect. However, it is one of the most important pieces in a successful Microservice deployment.

Going back to the list of principles we highlighted before. Let us outline the essential characteristics of an effective Microservice implementation:

- **Divided by business function:** This logical partitioning creates self-contained "products" for each Microservice, allowing the responsible team to make necessary changes without external dependencies.
- **Automation through CI/CD:** As discussed in *Chapter 6, Testing and CI*, CI/CD is crucial for maintaining the benefits of Microservices and streamlining updates.
- **Independent deployment:** Achieved through tools such as Kubernetes and **Infrastructure as Code** (**IaC**) solutions, as explored in *Chapter 6, Testing and CI*, autonomous deployment ensures that changes to one Microservice only trigger deployment for that

specific service.

- **Encapsulation:** Microservices should conceal their internal workings, acting as individual products that expose APIs for other services. This can be achieved using REST interfaces or messaging middleware, and enhanced with API Gateways.
- **Decentralized with no single point of failure:** To prevent the spread of failure, Microservices must be designed with no single point of failure. We will explore discovery services in the context of service meshes later.
- **Failures should be isolated:** To prevent cascading issues, each Microservice should be equipped to handle its own data and use circuit breakers to isolate failures. This approach can result in multiple independent databases, which may present challenges.
- **Observable:** As teams deploy changes automatically, robust observability is critical for managing large-scale failures and maintaining overall system health.

If we do not satisfy these requirements, then we are creating small Monoliths. A Monolith is great; it might be the best solution for your application. However, splitting it up requires more than separating the various pieces into smaller groups.

There is a lot to discuss but in the next few sections, we will cover some of the big pieces as they apply to the world of Spring and Java.

Service mesh

A service mesh is a dedicated infrastructure layer for managing communication between Microservices in a distributed system. It provides a way to control how different parts of an application interact with each other, enabling developers to manage complex distributed systems more easily.

The core principles of a service mesh typically include the following:

- **Service discovery and routing:** The service mesh provides a centralized registry of services and their locations, and it can automatically route requests between services based on configurable policies.
- **Load balancing:** The service mesh can distribute incoming traffic across multiple instances of a service, ensuring that each instance is used efficiently and that the system as a whole can handle high volumes of traffic.
- **Security:** The service mesh can enforce security policies such as authentication and authorization, and it can encrypt communication

between services to ensure that sensitive data is protected. Furthermore, it can verify the identity of a service, thus preventing a rogue deployment.

- **Observability:** The service mesh provides detailed metrics and logs that enable developers to monitor and debug the system more easily, and it can automatically generate performance and error reports.
- **Traffic control:** The service mesh can implement traffic-shaping policies that allow developers to control the flow of traffic between services, including rate limiting and circuit breaking.

Service mesh implementations work similarly to one another, as shown in *figure 8.1*. The service mesh uses a proxy instead of a direct connection between the various services. This is very similar to the Spring proxy architecture, only this is implemented externally to the application:

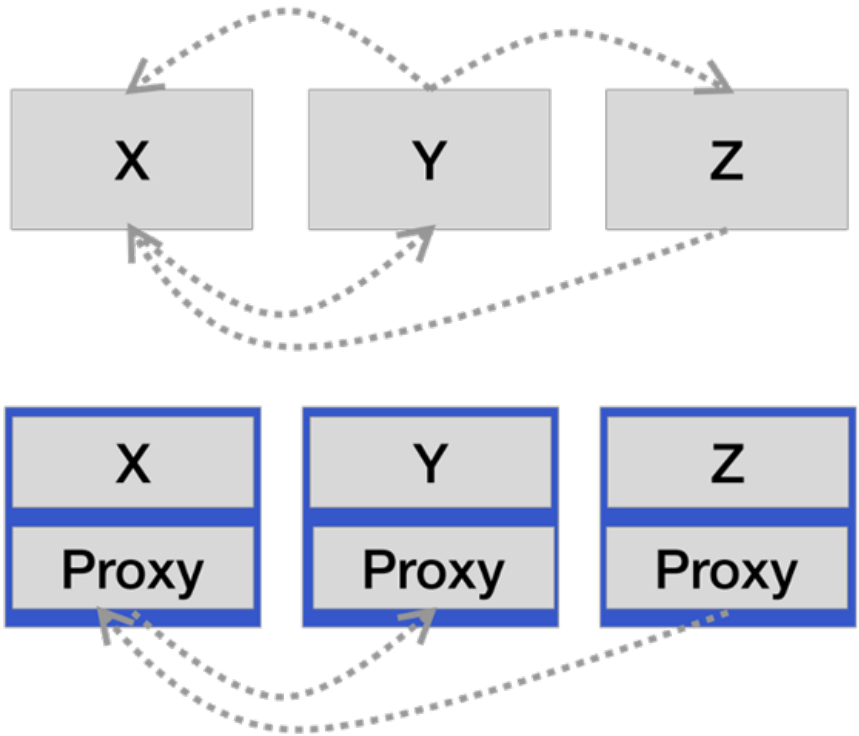

Figure 8.1: *Simplified service mesh implementation*

The proxy of a Service Mesh can do the following:

- **Discovery:** It can locate the right service that implements the functionality we require.
- **Load balancing:** It can work as a load balancer when we have multiple instances of a service.

- **Traffic splitting:** When we release a new version of the service, we can test it on a small segment of the traffic (Canary release).
- **Security:** It encrypts communication between the services.
- **Authentication:** It can validate that the server on the other side is one of our servers using a certificate.
- **Circuit breaker:** It can stop communication in case of a failure to prevent overhead.
- **Observability:** Since the proxy is at a central point and the mesh has a unique view of the entire system, it can seamlessly add observability information.

This is mostly seamless for the application layer. We can keep the code unchanged while using a typical service mesh. Configuration of the mesh is a part of the deployment process which provides a deep level of flexibility. There are several popular service mesh implementations, including Istio, Linkerd, Consul, and Envoy. Each implementation has its own strengths and weaknesses, and the best choice depends on the specific needs of the application. Here is a brief overview of some of the key differences between these service mesh implementations:

- **Istio**[3]**:** Istio is one of the most widely used service meshes and has strong support for Kubernetes. It offers powerful features for traffic management, security, and observability, including powerful routing rules, mutual TLS encryption, and integration with popular monitoring tools such as Prometheus and Grafana. However, Istio can be complex to configure and may have a higher resource overhead than some other service meshes.
- **Linkerd:** Linkerd is a lightweight service mesh that focuses on simplicity and ease of use. It provides strong support for Kubernetes and offers features such as automatic mTLS, request tracing, and canary deployments. However, it may lack some of the advanced features of other service meshes, and its community and ecosystem may not be as large as Istio's.
- **Consul:** Consul is a service mesh that is tightly integrated with HashiCorp's broader suite of tools for infrastructure automation. It provides features for service discovery, routing, and security, and also includes a powerful key-value store that can be used to manage configuration data. However, its feature set may be more limited than some other service meshes, and it may be less widely adopted.
- **Envoy:** Envoy is a proxy server that is often used as a data plane for service meshes such as Istio and Linkerd. It provides powerful features for traffic management, including load balancing, circuit breaking, and advanced routing rules. However, it requires more

configuration and management than some other service meshes, and may not provide as much out-of-the-box functionality.

For simple cases, the overhead and logistics of a service mesh make it impractical. However, as the number of Microservices rises the need for a solution like this grows rapidly. *Figure 8.2* shows the Istio Kiali dashboard in action:

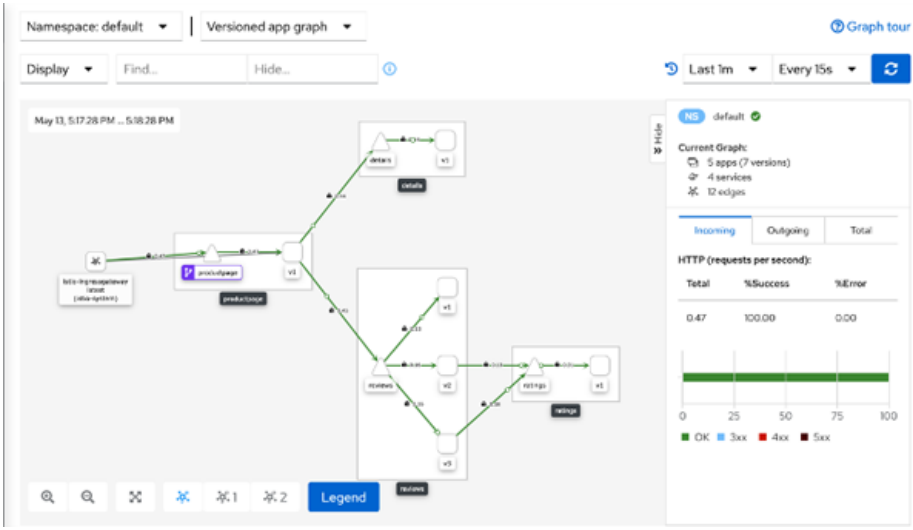

Figure 8.2: *Istio Kiali dashboard*

Authentication and authorization

Identity is the biggest challenge we face when transitioning to a microservice architecture. In the past, it was enough to authenticate users against a table in the database, but this is not an option when Microservices are involved. Let us start with an example; say we have an HR system, and we want to get a list of employees. In a Monolith, we would do the following:

- Verify the user is logged in and has manager privileges.
- Return employee list.

This is trivial and should work reasonably well. A naïve view of Microservices might assume that every service manages its own users; this will not scale. We would have no way of managing or communicating the changes. Thankfully, there are common standards for authentication that let us communicate the user list all around. The **JSON Web Token** (**JWT**) is best known, although **Platform Agnostic Security Token** (**PASETO**) and others are gaining some traction. With a token, an independent service can verify a request is valid without an additional connection. Look closely at *figure 8.3*. The important part of that figure is the part that is missing

from the figure.

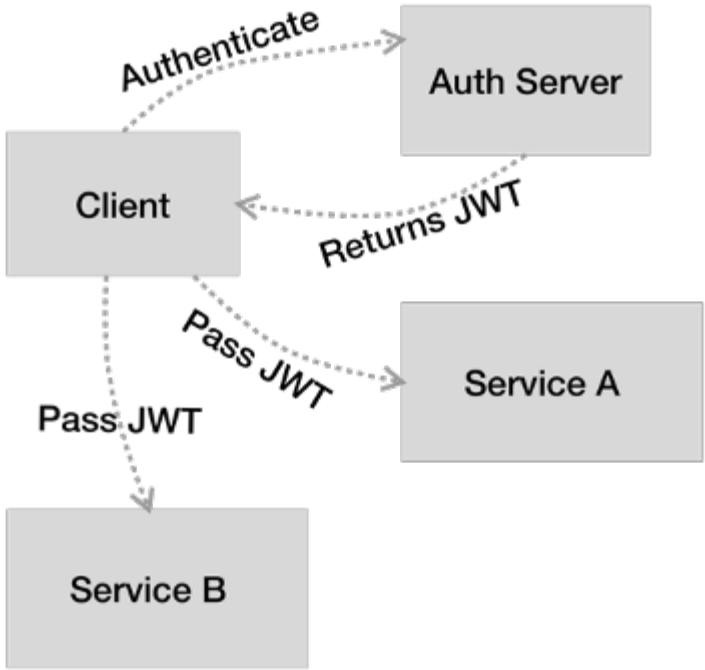

Figure 8.3: *Client using JWT, can you Spot the Missing Part?*

The missing part is testing the JWT token. Notice that both Service A and Service B accept the token, but there is no connection to the Auth Server to verify that this is indeed a valid token. The reason is hidden in the JWT structure, it consists of the following three parts:

1. **Header:** This identifies that this is a JWT, and the algorithm used.
2. **Payload:** Here, we can place information that we need to carry everywhere, containing important details about the user who has this token. It can tell us the role of the user which would be important in this case (more on that soon).
3. **Signature:** This is a digital signature that shows us the JWT is authentic and was generated by our authorization server. By verifying the signature and the payload, we know that the JWT is valid. There is no need to check with the authorization server.

Notice that we discussed the “manager privileges” before. That indicates role-based authorization. Authentication means we know who you are. Authorization means we know what you are allowed to do in the system. Users can be assigned multiple roles and with **Role Based Access Control** (**RBAC**), we can block users who do not have the right permissions.

User authentication, registration, management, and authorization are relatively generic concepts. That is where multiple services in the cloud and on-premise come into play; they encapsulate all of these pieces under a single manageable service.

There are several common authentication solutions for Java developers; Keycloak[4] is probably the best-known among the open-source options. There are many cloud authentication providers such as Okta, Auth0 (owned by Okta), FusionAuth, and so on. A more recent contender is the Spring Authorization Server[5], which is a relatively new project and not as functional as Keycloak at the time of this writing. *Figure 8.4* shows the Keycloak user management user interface:

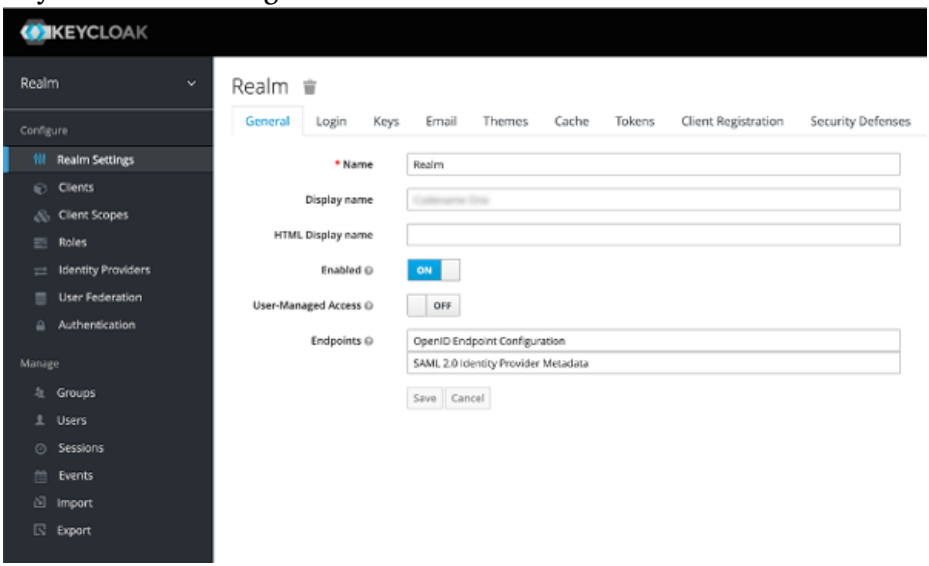

Figure 8.4: *Keycloak user management UI*

It is easiest to set up Keycloak with a container image. Once the server is running it can be integrated into a Spring Boot application with a dependency:

1. <dependency>
2. <groupId>org.keycloak</groupId>
3. <artifactId>keycloak-spring-boot-starter</artifactId>
4. </dependency>

We can enable Keycloak using a configuration definition that implicitly forces a secure login through the application:

1. @KeycloakConfiguration
2. public class SecurityConfiguration extends KeycloakWebSecurityConfigurerAdapter {

```
3.
4. @Autowired
5. public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
6. SimpleAuthorityMapper simpleAuthorityMapper = new SimpleAuthorityMapper();
7. simpleAuthorityMapper.setPrefix("ROLE_");
8.
9. KeycloakAuthenticationProvider keycloakAuthenticationProvider
10. = keycloakAuthenticationProvider();
11. keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(simpleAut
12. auth.authenticationProvider(keycloakAuthenticationProvider);
13. }
14.
15. @Bean
16. public KeycloakSpringBootConfigResolver KeycloakConfigResolver() {
17. return new KeycloakSpringBootConfigResolver();
18. }
19.
20. @Bean
21. @Override
22. @ConditionalOnMissingBean(HttpSessionManager.class)
23. protected HttpSessionManager httpSessionManager() {
24. return new HttpSessionManager();
25. }
26.
27. @Bean
28. @Override
29. protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
30. return new RegisterSessionAuthenticationStrategy(
31. new SessionRegistryImpl());
32. }
33.
34. @Override
35. protected void configure(HttpSecurity httpSecurity) throws Exception {
```

```
36. super.configure(httpSecurity);
37. httpSecurity.csrf()
38. .disable()
39. .authorizeRequests()
40. .antMatchers("/appsec/**")
41. .authenticated()
42. .anyRequest()
43. .permitAll();
44. httpSecurity.headers().
45. addHeaderWriter(new
    XFrameOptionsHeaderWriter(XFrameOptionsHeaderWriter.XFrameOptionsMode.S
46. }
47. }
```

The **configureGlobal** method helps us to map roles as they are defined in Keycloak to role definitions within Spring Boot. Once those are mapped, we can use Spring Boot role annotations to restrict access on a class or method level in the code based on roles. Roles can be defined in Keycloak via the administrative user interface or via the API.

The **@Bean** annotated methods provide Keycloak implementations for various Spring Boot APIs. They will be injected instead of the default implementation. The final configure call is used to define the security configuration. The main point to notice here is the **antMatcher** which we use to apply security restrictions to a specific area of the site.

An Ant-style path pattern is a string that can contain the ? and * characters as wildcards. The “?” character matches any single character, whereas the “*” character matches zero or more characters. For example, the **pattern /admin/**** would match any URL that starts with **/admin/**, regardless of the number of additional path segments. In Spring Security, AntMatcher is often used in conjunction with the HttpSecurity class to define the security configuration for an application.

Eventual consistency

Ideally, services should be stateless, as managing the state introduces added complexity. Storing state in the client requires constant transmission between client and server, whereas server-side storage demands frequent retrieval, caching, or local storage, potentially hindering scalability.

A typical Microservice uses its own database and processes local data. When remote data is required, caching is often used to minimize

communication with remote services, contributing to the scalability of Microservices. In a monolithic architecture, the database often becomes a bottleneck, with efficiency constrained by data storage and retrieval speeds. This limitation presents the following two significant challenges:

1. **Size:** As the volume of data increases, the database size expands, impacting the performance of all users simultaneously. For example, consider searching an SQL table containing every Amazon purchase ever made to locate a specific transaction.
2. **Domain:** Different databases cater to various use cases, with optimizations for consistency, read or write speeds, temporal or spatial data, and more. A user-tracking Microservice might use a time-series database, optimized for time-related data, whereas a purchase service may rely on a traditional, conservative ACID[6] database.

Imagine a common scenario, where a user performs a purchase, and it seems to go through, but billing fails. In a Monolith, we would rollback the database transaction and restore the database to the previous state. This is good but has the following drawbacks:

- A transaction can block for a long time.
- Distributed transactions can be problematic at the cloud scale.

For Microservices, this is not an option; their answer to that is the Saga pattern. It is a design pattern for managing distributed transactions in a Microservices architecture. It is used to ensure that a series of transactions that need to be executed in a distributed environment are executed in a consistent and reliable manner. The Saga pattern is based on the idea of breaking down a long-running transaction into a series of smaller, more manageable transactions called “sagas”. Each saga is responsible for executing a specific portion of the overall transaction.

The Saga pattern uses compensating transactions to reverse the impact of a saga in case of failure. If a saga encounters a failure, the system executes the compensating transaction to revert the modifications made by the preceding transaction, enabling the system to recover from failures and preserve consistency.

In contrast to ACID, the **BASE** principle, which essentially stands for **Basically Available, Soft state, Eventual consistency**, is used in distributed systems to provide high availability and scalability at the cost of consistency. BASE transactions are not atomic and may leave the system in an inconsistent state temporarily, but they guarantee **eventual consistency**. In effect, the BASE gives up on the Consistency and Isolation aspects of ACID.

In *figure 8.5*, we can see a simple Saga flow in broad terms:

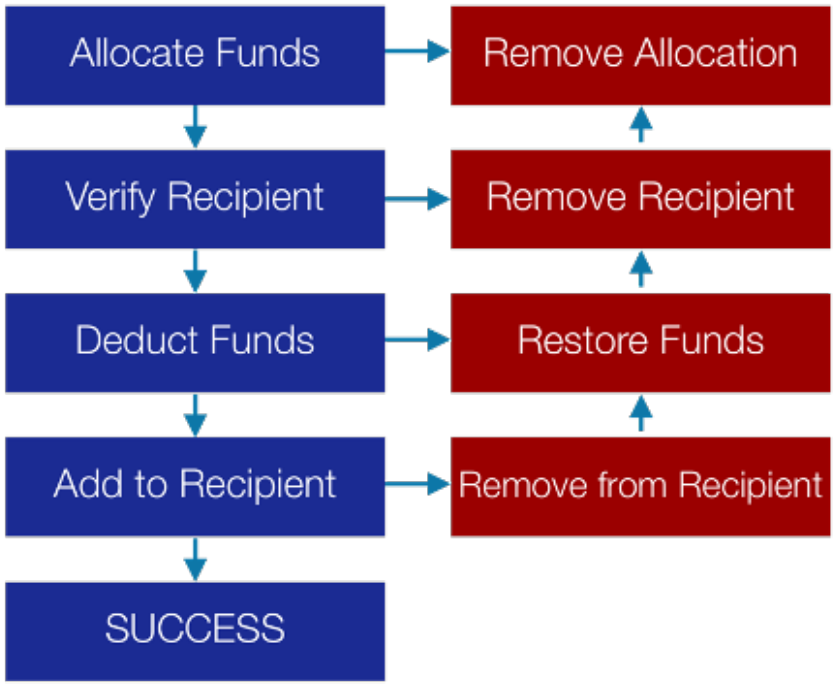

Figure 8.5: *Saga pattern sample*

This flow demonstrates a financial transfer process with the following steps:

1. Begin by allocating funds.
2. Next, confirm the validity and existence of the recipient.
3. Proceed to withdraw the funds from the sender's account.
4. Finally, deposit the money into the recipient's account.

This sequence constitutes a successful transaction. In a conventional database, this would be a single transaction, as illustrated in the blue column on the left. However, if an error occurs, we must execute the inverse process:

- If fund allocation fails, remove the allocation by implementing a separate block of code to perform the opposite action.
- If recipient verification fails, remove the recipient and reverse the allocation.
- If fund deduction encounters an issue, restore the funds, remove the recipient, and reverse the allocation.
- Finally, if transferring funds to the recipient's account is

unsuccessful, execute all the compensating actions to undo the transaction.

Saga aligns nicely with the CAP theorem, which is a fundamental concept in distributed computing, that states that in a distributed system, it is impossible to guarantee all three of the following properties at the same time: consistency, availability, and partition tolerance.

Consistency refers to the idea that all nodes in a distributed system see the same data at the same time. **Availability** refers to the ability of a system to continue operating and responding to requests even in the presence of failures. **Partition tolerance** refers to the ability of a distributed system to continue functioning even if network partitions occur, which means that communication between some nodes is lost.

The CAP theorem states that a distributed system can only provide two out of the three properties at the same time. For example, a distributed system can choose to provide consistency and partition tolerance, but at the cost of availability. Alternatively, a system can choose to provide availability and partition tolerance, but at the cost of consistency.

In practical terms, this means that when designing a distributed system, architects and developers need to carefully consider which two properties are most important for their particular use cases. In some cases, consistency may be the top priority, whereas in other cases, availability may be more important.

It is important to note that the CAP theorem does not imply that distributed systems must sacrifice one of these properties entirely, but rather that they cannot guarantee all three simultaneously in the event of a network partition or other failure. In *figure 8.6*, we can see a common representation of the tradeoffs in the CAP Theorem as a Venn diagram:

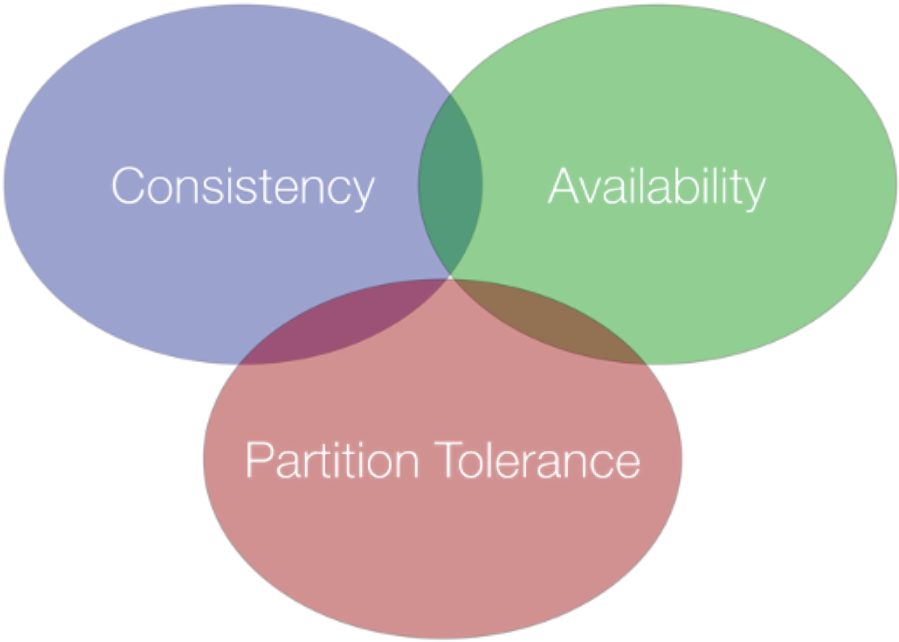

Figure 8.6: *The tradeoffs in CAP theorem*

Saga choreography and orchestration are two different approaches to implementing the Saga pattern for managing distributed transactions in a Microservices architecture.

When using choreography, each service involved in the transaction communicates with other services directly to coordinate the transaction. Each service initiates and participates in transactions by exchanging messages with other services. Each service knows which other services it needs to communicate with and in what order, and they work together to ensure that the transaction is executed correctly.

When using orchestration, a centralized component called the orchestrator is responsible for coordinating the transaction. The orchestrator sends messages to each service to initiate the transaction, and it also specifies the order in which each service should execute its portion of the transaction. The orchestrator monitors the progress of each service and sends compensating transactions, if necessary, to undo any changes made by services that failed to complete their portion of the transaction.

The main difference between the two approaches is the degree of centralization. In Saga choreography, there is no central coordinator, and each service communicates with other services directly. This approach is more decentralized and can be more flexible, but it can also be more complex to implement and manage.

When using orchestration, there is a central orchestrator that manages the entire transaction. This approach is more centralized and can be easier to manage, but it can also be less flexible and less tolerant of failure.

The choice between Saga choreography and orchestration depends on the specific requirements of the system being developed. If the system is highly decentralized and requires a high degree of flexibility, Saga choreography may be a better fit. If the system is more centralized and requires tighter control over the transaction, Saga orchestration may be a better fit. For a small number of services, choreography can work but it gets difficult as keeping track becomes challenging.

This is a big subject and there are a few orchestration solutions for Java. The best-known are probably Apache Camel and Camunda. We can integrate both with Spring Boot, for example, the following dependency should add Apache Camel to a Spring Boot application:

```
<dependency>
<groupId>org.apache.camel.springboot</groupId>
<artifactId>camel-spring-boot-starter</artifactId>
<version>${camel.version}</version>
</dependency>

<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jdbc</artifactId>
<version>${camel.version}</version>
</dependency>

<dependency>
<groupId>org.apache.camel.springboot</groupId>
<artifactId>camel-jdbc-starter</artifactId>
<version>${camel.version}</version>
</dependency>
```

When creating a Saga, we define the endpoints that implement the various parts and the compensating actions for each stage. This means that if the Saga fails, Apache Camel will invoke compensating operations to restore the application back to a consistent state:

```
from("direct:newOrder")
.saga()
```

3. .propagation(SagaPropagation.MANDATORY)
4. .compensation("direct:cancelOrder")
5. .transform().header(Exchange.SAGA_LONG_RUNNING_ACTION)
6. .bean(orderManagerService, "newOrder")
7. .log("Order ${body} created");

Messaging

A messaging system, also known as a message-oriented middleware or message broker, is a software component that facilitates asynchronous communication between distributed components or applications by exchanging messages. It acts as an intermediary, enabling the transfer of messages between producers (senders) and consumers (receivers) without requiring them to be aware of each other's existence or implementation details.

Messaging is essential in a Microservices architecture for several reasons, including the following:

- **Decoupling:** Microservices are designed to be small, focused, and loosely-coupled components that can evolve independently. Messaging enables asynchronous communication between Microservices, allowing them to interact without being tightly bound to each other. This decoupling facilitates independent scaling, deployment, and modification of individual services without impacting the entire system.
- **Resilience:** In a Microservices architecture, multiple services work together to fulfill complex business requirements. Messaging systems provide a buffer to handle temporary failures, ensuring that messages are not lost during service disruptions. This contributes to the overall resiliency and fault tolerance of the system.
- **Scalability:** Messaging systems enable horizontal scaling by supporting the distribution of messages across multiple instances of a service. This allows a system to handle increased loads by adding more instances of the required services. Furthermore, messaging systems can handle load balancing and message routing to efficiently distribute messages across the available service instances.
- **Event-driven architecture:** Messaging facilitates event-driven architecture, where services react to events generated by other services or external systems. This promotes real-time processing, better responsiveness, and the ability to adapt to changing business requirements.
- **Load leveling:** Messaging systems can act as a buffer to absorb fluctuations in the demand for a particular service, enabling load

leveling. This helps to maintain the stability of the system under varying workloads, preventing individual services from becoming overwhelmed.

- **Data consistency:** In distributed systems, ensuring data consistency across multiple services can be challenging. Messaging helps to maintain consistency by providing a means to propagate changes, such as updates or deletions, across different services. By using messaging patterns such as event sourcing and the Saga pattern, Microservices can coordinate transactions and maintain data consistency across distributed systems.
- **Flexibility and adaptability:** Messaging allows Microservices to communicate using common messaging protocols and formats. This flexibility enables the system to adapt to changing requirements, as new services can be introduced, or existing services can be modified without disrupting the entire architecture.
- **Monitoring and observability:** Messaging systems can provide valuable insights into the performance and health of the Microservices ecosystem. By monitoring message queues, topics, and throughput, developers can gain a better understanding of the system's performance, identify bottlenecks, and detect potential issues early on.

There are several popular message brokers available for Spring developers, and the most common are as follows: RabbitMQ, Apache Kafka, and Spring Cloud Stream.

RabbitMQ

RabbitMQ is an open-source message broker that implements the **Advanced Message Queuing Protocol** (**AMQP**). It provides a highly available, scalable, and fault-tolerant messaging system for Spring Boot Microservices.

Spring Boot provides excellent support for RabbitMQ through the **spring-boot-starter-amqp** module, which includes the RabbitTemplate for sending and receiving messages and the **@RabbitListener** annotation for message-driven POJOs.

Key RabbitMQ features include the following:

- Support for different messaging patterns such as publish/subscribe, request/reply, and point-to-point.
- Highly available and fault-tolerant through clustering and mirrored queues.
- Easy integration with Spring Boot using the **spring-boot-starter-**

amqp module.

Apache Kafka

Apache Kafka is a distributed streaming platform that excels in high-throughput, fault-tolerant, and scalable messaging for Microservices. It is suitable for real-time data processing, streaming, and event-driven architectures. It is the leader in the field and provides many important features beyond messaging.

Spring Boot supports Kafka through the **spring-boot-starter-kafka** module. This module provides the **KafkaTemplate** for sending messages, **KafkaListener** for consuming messages, and support for **KafkaStreams** for stream processing.

Stream processing is a computational paradigm that focuses on the continuous processing and analysis of data streams in real-time. In contrast to batch processing, which processes data in large chunks at specific intervals, stream processing deals with data as it is generated, often from multiple sources such as sensors, user interactions, or other applications. The primary goal of stream processing is to extract valuable insights, detect patterns, or trigger actions based on incoming data without significant delay.

Key Apache Kafka features include the following:

- High-throughput and low-latency messaging for large-scale, distributed systems.
- Log-based storage system that ensures strong durability and data retention.
- Support for stream processing through Kafka Streams API.
- Easy integration with Spring Boot using the **spring-boot-starter-kafka** module.

Spring cloud stream

Spring Cloud Stream is a framework built on top of Spring Boot and Spring Integration that simplifies the development of event-driven, message-driven Microservices. It provides a programming model for building applications that can communicate with external messaging systems such as RabbitMQ, Apache Kafka, and others.

Spring Cloud Stream offers a declarative approach to configure message channels, allowing developers to focus on the business logic while the framework handles the communication. It also supports a variety of binder implementations for different message brokers, making it easy to switch

between them without changing the application code.

Key Spring Cloud Stream features include the following:

- Abstracts the messaging infrastructure, allowing developers to focus on business logic.
- A declarative approach to configure input and output channels.
- Supports multiple binder implementations for different messaging platforms.
- Built-in support for content-type negotiation and message conversion.

Publish subscribe and point to point

Message brokers facilitate communication between distributed applications or components by managing message exchange. Two common messaging patterns used in message broker interactions are the publish-subscribe pattern and message queue pattern (also known as point-to-point). Each pattern has distinct characteristics and is suited to different scenarios:

- **Publish-subscribe pattern**: In the publish-subscribe (pub-sub) pattern, messages are sent to "topics" by publishers (producers) and are then broadcasted to all subscribers (consumers) that have expressed interest in that specific topic. This pattern allows for one-to-many communication where one publisher can send messages to multiple subscribers simultaneously.
- **Message queue pattern**: Here, messages are sent to a "queue" by producers and are then consumed by one or more consumers. Each message in the queue is processed by only one consumer, even when multiple consumers are listening to the same queue. This pattern enables point-to-point communication between individual components or applications.

The key characteristics of the publish-subscribe pattern include the following:

- **Decoupling:** Publishers and subscribers are loosely coupled, as they are not directly aware of each other's existence.
- **Fan-out:** A single message published on a topic can be received by multiple subscribers, enabling efficient dissemination of information.
- **Dynamic subscription:** Subscribers can join or leave topics at any time, allowing for flexible and dynamic communication.
- **Event-driven:** The publish-subscribe pattern is well-suited for event-driven architectures, where components react to events

generated by other components or external systems.

In *figure 8.7* we see the difference between publish-subscribe (top) and point-to-point messaging.

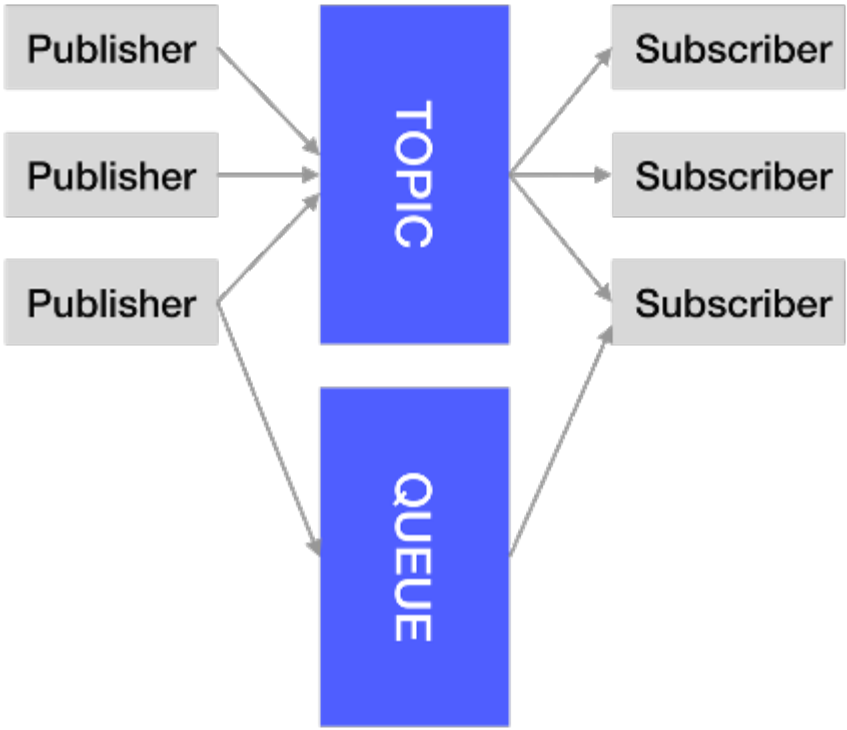

Figure 8.7: *Publish-subscribe versus point-to-point messaging*

Typical uses for publish-subscribe include the following:

- News feeds
- Real-time analytics
- Stock market updates
- Notification system
- Chat and collaboration
- Distributed monitoring
- Multiplayer gaming
- Content recommendation, and much more.

Every system in which multiple independent systems can respond to an action can potentially relate to such a system.

To test messaging in action, we first need the message broker. In this case, we can use RabbitMQ which can be installed with the following

commands:

1. docker pull rabbitmq:3-management
2. docker run -d --hostname my-rabbit --name rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq:3-management

This will launch RabbitMQ within a Docker container and will make the management console available under **http://localhost:15672/** with the username "**guest**" and password "**guest**". Once this is done, we can create a Spring Boot application to implement a simple queue.

The bare minimum Spring initializer project is shown in *figure 8.8*:

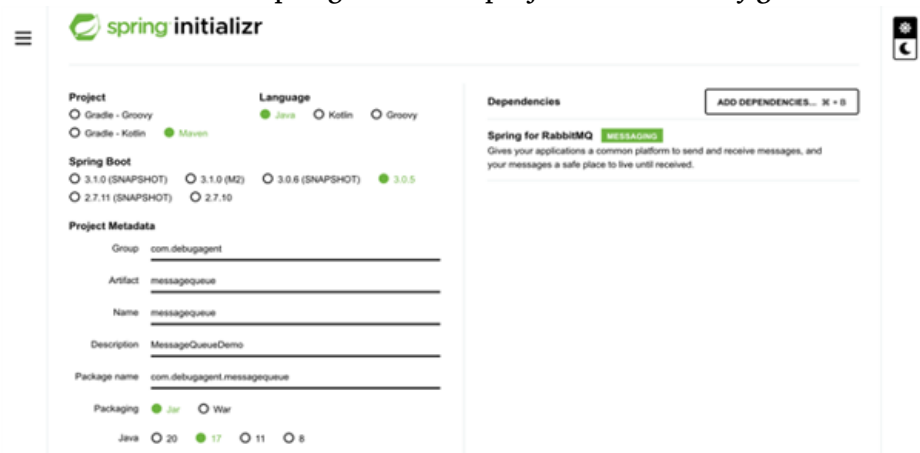

Figure 8.8: *Minimal Spring Initializr Project for RabbitMQ*

Once we have that project, we first need to configure the settings that connect Spring Boot to RabbitMQ in the **application.properties** file:

1. spring.rabbitmq.host = localhost
2. spring.rabbitmq.port = 5672
3. spring.rabbitmq.username = guest
4. spring.rabbitmq.password = guest

Once we set those properties, we can add a configuration for the queues. In this case, we are adding one queue:

```
1. @Configuration
2. public class RabbitMqConfig {
3.
4. public static final String TOPIC_EXCHANGE_NAME = "example-topic-exchange";
5. public static final String QUEUE_NAME_1 = "example-queue-1";
6.
```

```
@Bean
public TopicExchange topicExchange() {
return new TopicExchange(TOPIC_EXCHANGE_NAME);
}

@Bean
public Queue queue1() {
return new Queue(QUEUE_NAME_1, false);
}

@Bean
public Binding binding1(TopicExchange topicExchange, Queue queue1) {
return BindingBuilder.bind(queue1).to(topicExchange).with("example.routingkey.1");
}
}
```

The key part of the message publisher is the publish call, which invokes the **convertAndSend** API on the rabbit template. Message systems are platform agnostic and convert the objects we send into a serialized format, such as JSON, for delivery:

```
@Component
public class MessagePublisher {

private final RabbitTemplate rabbitTemplate;
private final TopicExchange topicExchange;

public MessagePublisher(RabbitTemplate rabbitTemplate, TopicExchange topicExchange) {
this.rabbitTemplate = rabbitTemplate;
this.topicExchange = topicExchange;
}

public void publish(String routingKey, String message) {
rabbitTemplate.convertAndSend(topicExchange.getName(), routingKey, message);
```

14. }
15. }

The subscriber is much simpler; it mostly consists of an annotation denoting the callback method:

```
1. @Component
2. public class MessageSubscriber {
3. @RabbitListener(queues = RabbitMqConfig.QUEUE_NAME_1)
4. public void handleMessage(String message) {
5. System.out.println("Subscriber received message: " + message);
6. }
7. }
```

Once this is done, we can invoke the publish method to send a push to the subscribers:

```
1. messagePublisher.publish("example.routingkey.1", "Message for Subscriber");
```

This works as expected locally but also scales nicely when sending messages between datacenters. In *figure 8.9*, we can see the RabbitMQ management console Web interface. Through here we can observe the load on the system and troubleshoot messaging problems:

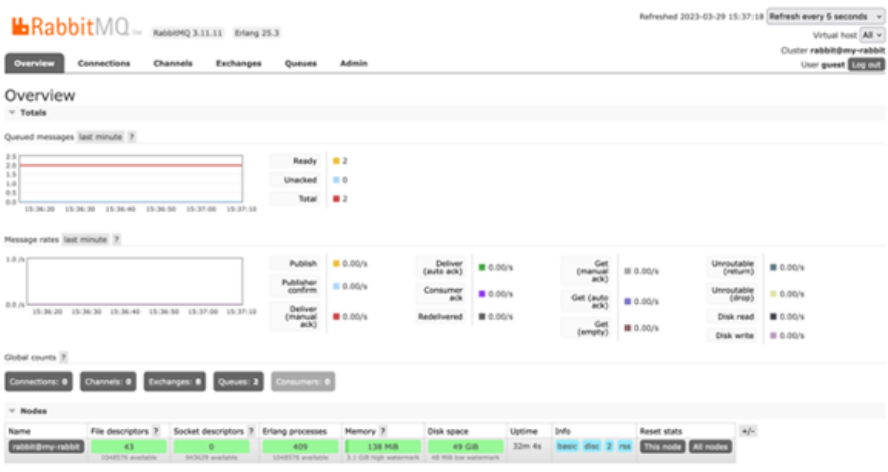

Figure 8.9: *The RabbitMQ management console*

Monolith first

The prevalent industry view suggests that projects should generally begin as monoliths and transition to Microservices only if necessary. By now, you should recognize the challenges associated with deploying Microservices correctly, as well as the significant trade-offs required to enable scalability.

While Microservices do present drawbacks compared to Monoliths, they offer valuable benefits that should not be overlooked.

Microservices represent a balance between the developer and DevOps expertise. The complexity of a Monolith is shifted to the responsibilities of the DevOps team. While this may be reasonable, limited DevOps resources can make adopting Microservices inadvisable. A capable DevOps team is essential for considering Microservices; otherwise, companies may devote more time to operational complexities than code development.

The most significant advantage of Microservices is their impact on the team. Establishing stable teams with well-defined scopes is vital. Creating independent vertical teams offers substantial benefits. Even the most modular Monolith cannot compete with a well-planned microservice deployment in terms of team scalability. With hundreds of developers, tracking git commits and code changes at scale becomes unmanageable, and the true value of Microservices is realized only in large teams.

This principle may seem reasonable, but startups often experience sudden shifts. In such environments, downsizing can quickly transform a mid-sized company into a small team. Startups that have adopted a Microservice architecture may face difficulties maintaining numerous services in various languages after downsizing. Determining the precise size at which a Monolith becomes impractical is challenging, but consider the following questions:

- **How many teams are currently in place?** A Monolith may be suitable for a few teams, while a dozen teams might encounter issues with this approach.
- **Evaluate pull requests and issue resolution durations.** As a project expands, pull requests will take longer to merge, and issue resolutions will be delayed. This is a natural consequence of increasing complexity. Keep in mind that new projects may include larger features that could skew initial results. Once these factors are considered, a measurable decrease in productivity should become evident.

Note that this is only one metric, and it may indicate other areas needing attention, such as test pipeline optimization, review processes, or modularity.

- **Are there knowledgeable experts familiar with the code?** As a project grows, experts may lose track of finer details, leading to challenges when addressing bugs or making decisions without consultation.
- **Are you willing to allocate additional resources?** Implementing

Microservices tends to be more expensive, and while there may be instances where scale can be fine-tuned, costs associated with observability and management often outweigh potential savings. However, since personnel expenses typically surpass cloud hosting costs, the overall balance may still favor Microservices in cases of significant scaling.

The trade-offs of Monolith versus Microservice are illustrated nicely in the radar chart in *figure 8.10*. Notice that this chart was designed with a large project in mind. The smaller the project, the better the picture is for the Monolith:

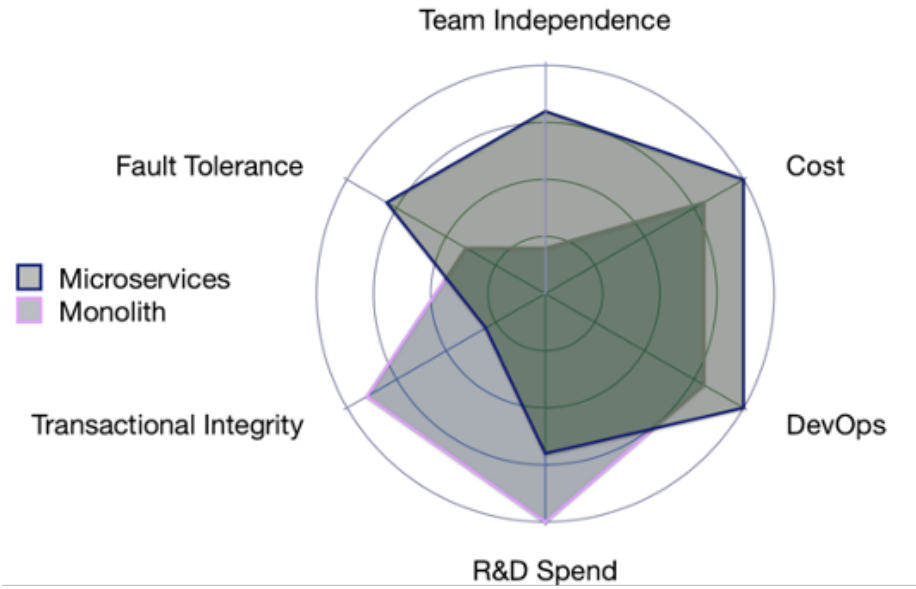

Figure 8.10: *Tradeoffs of Microservices versus Monolith*

Modular Monolith

When envisioning a Monolith, many of us recall the massive, unwieldy codebases that companies like Amazon and eBay replaced. These immense codebases made every modification cumbersome, and scaling them posed significant challenges. There are even accounts of enormous binaries requiring specialized infrastructure for deployment and files exceeding compiler-supported sizes.

However, comparing these examples to Microservices is not entirely fair. Newer approaches tend to outperform their predecessors. A more equitable comparison would involve evaluating Microservices against a contemporary Monolith built using modern tooling. In such scenarios, the Monolith can offer numerous advantages.

To better understand this comparison, we must consider the limitations of modern Monoliths and explore what a contemporary Monolith might look like.

Modulith

To gain insight into a modern Monolith, consider the Spring Modulith project. This modular Monolith enables the construction of a Monolith with dynamically isolated components. This approach facilitates the separation of testing, development, documentation, and dependencies, offering some of the isolation benefits of microservice development with minimal overhead. It eliminates the need for remote calls and duplication of functionalities, such as storage and authentication.

Spring Modulith does not depend on Java platform modularization (Jigsaw) but enforces separation during testing. At runtime, the Modulith functions as a standard Spring Boot project, with some additional runtime capabilities for modular observability. Primarily, it serves as a best practices enforcer. The benefits of this separation extend beyond typical Microservices experiences, though there are trade-offs. For example, a conventional Spring Monolith would exhibit a layered architecture with packages, such as:

1. com.debugagent.theapp
2. com.debugagent.theapp.services
3. com.debugagent.theapp.db
4. com.debugagent.theapp.rest

This approach is beneficial as it helps prevent dependencies between layers. For instance, the database layer should not depend on the service layer. Using modules in this manner enforces a downward dependency graph. Although this might be suitable for smaller projects, it becomes less practical as the project expands. As the project grows, each layer accumulates business logic classes and database complexities. With a Modulith, the architecture would resemble the following:

1. com.debugagent.theapp.customers
2. com.debugagent.theapp.customers.services
3. com.debugagent.theapp.customers.db
4. com.debugagent.theapp.customers.rest
5.
6. com.debugagent.theapp.invoicing
7. com.debugagent.theapp.invoicing.services
8. com.debugagent.theapp.invoicing.db
9. com.debugagent.theapp.invoicing.rest

10.
11. com.debugagent.theapp.hr
12. com.debugagent.theapp.hr.services
13. com.debugagent.theapp.hr.db
14. com.debugagent.theapp.hr.rest

This resembles a well-structured microservice architecture, with components separated based on business logic. Cross dependencies are more effectively managed, allowing teams to focus on their respective areas without interfering with one another. This achieves many of the benefits of Microservices without the associated overhead.

Separation can be further enforced using annotations to define and maintain one-way dependencies between modules. For example, the human resources module would not be related to invoicing, nor would the customer module. A one-way relationship can be established between customers and invoicing, with communication facilitated by events. Within a Modulith, events are straightforward, fast, and transactional, decoupling dependencies between modules without complications. Enforcing this level of separation in Microservices can be challenging. For instance, if invoicing needs to expose an interface to another module, how can you prevent customers from accessing it?

With modules, this control is possible. Although a user could modify the code to grant access, such changes would need to undergo a code review, potentially raising issues. Modules can still incorporate common microservice elements, such as feature flags and messaging systems.

In a module system, every dependency is explicitly mapped out and documented within the code. The Spring implementation offers automatic documentation, generating up-to-date charts for easy reference.

Other modules

Standard Java Platform Modules (Jigsaw) can be employed to construct a Spring Boot application. This approach benefits from breaking down the application using standard Java syntax, although it may occasionally prove cumbersome. This method is most effective when working with external libraries or dividing tasks into shared tools.

Another alternative is using Maven's module system, which allows the build to be separated into multiple distinct projects. This process is highly convenient, preventing the challenges associated with massive projects. Each project remains self-contained and easily manageable, complete with its own build process. As the main project is built, everything comes together to form a single Monolith. In many ways, this solution aligns with

the preferences of numerous developers.

The benefit

The ability to complete a transaction without relying on “eventual consistency” offers a significant advantage. Debugging and replicating a distributed system can be challenging, as intermediate states may be difficult to reproduce locally or fully comprehend through observability data.

Enhanced performance eliminates much of the network overhead. With properly optimized Level 2 caching, it is possible to reduce 80%–90% of read IO. While this can be achieved in a microservice architecture, it is more difficult to accomplish and may not eliminate network call overhead.

As previously noted, application complexity remains in a microservice architecture, but it is merely relocated. The addition of numerous moving parts increases overall complexity. Adopting a smarter and simpler unified architecture is a more sensible approach.

Conclusion

Microservices are far more complex than merely splitting up the Monolith. The Monolith itself is more than a giant block of code. To get the most out of both architectures, we need to go beyond the code and study the surrounding tooling. That means venturing into the realm of DevOps and structuring our applications for durability.

Points to remember

- Microservices are not small Monoliths.
- Eventual consistency makes scale possible but requires a great deal of work and carries a lot of runtime complexities.
- • The CAP theorem describes the tradeoffs between consistency, availability, and partition tolerance. You can have all three but in case of failure, you might have to compromise on two.
- Message Brokers help decouple and scale Microservices through asynchronous event-driven programming.

Multiple choice questions

1. What is CAP theorem?

a. Connected, Applied, Priorities

b. Consistency, Availability, Partition Tolerance

c. Collocated, Asynchronous, Packet Transmission

d. Changeable, Architecture, Presentation

e. Core, Avoidance, Process

2. In Saga pattern, we:

a. Have inconsistent/invalid state

b. Use compensating transactions to undo

c. Have multiple regular transactions

d. Orchestrate multiple small processes as if they were one

e. All of the above

3. A Modulith is:

a. A runtime module system

b. A generic term for modular Monolith

c. A Spring project that uses tests to enforce structure

d. A sub project of Jigsaw

4. Messages can be sent: (pick all that apply)

a. Point to point

b. To all nodes

c. Publish-subscribe

d. All nodes except X

Answers

1. b

2. e

3. c

4. a, c

---

1 https://www.martinfowler.com/articles/microservices.html

2 https://aws.amazon.com/microservices/

3 https://istio.io/latest/docs/setup/getting-started/

4 https://www.keycloak.org/

5 https://spring.io/projects/spring-authorization-server

6 ACID is a core concept of database design. It describes the properties of a reliable database and stands for Atomicity, Consistency, Isolation, and Durability.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 9
Serverless

Introduction

The complexities of Monoliths and Microservices pushed some developers toward an easier route. Serverless is a problematic branding term indicating that we no longer need to manage our server, but only the application. The serverless provider takes over the complexity of spinning up container instances for us as long as we follow some guidelines in application development.

It is a modern take on the concepts of **Platform as a Service** (**PaaS**), with a greater focus on horizontal scalability, statelessness, and cloud. The main power of serverless is in its connection to the surrounding cloud services, from which it can draw its true potential.

Structure

In this chapter, we will discuss the following topics:

- What is serverless
- Using AWS Lambda
- GraalVM and Monolith serverless
- The Cloud ecosystem

Objectives

In this chapter, we will learn about the benefits of serverless and the price we pay when picking it as our architecture of choice. We will learn how Java can be a better citizen in the world of serverless and the common pitfalls we might run into.

What is serverless

Serverless is an architectural approach and a computing paradigm in which developers build and run applications without having to manage the underlying infrastructure. In a serverless environment, cloud service providers dynamically allocate resources to execute the code as needed. The term “serverless” can be a bit misleading because there are still servers involved; however, the responsibilities of server management, capacity planning, and scaling are abstracted away from developers.

Serverless architecture typically revolves around **Function as a Service** (**FaaS**) platforms, which enable developers to write small, modular pieces of code called functions. These functions are designed to perform specific tasks and can be triggered by various events, such as an API call, a message in a queue, or a file being uploaded to a storage service.

Major cloud providers offer serverless platforms, such as AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions. These platforms take care of server provisioning, maintenance, and scaling, allowing developers to focus on writing code and implementing business logic.

The following table illustrates key aspects of serverless development:

	Traditional Server Development	Serverless Development
Cost efficiency	Never be fully utilized, leading to high costs	Can be cost-effective, but costs can skyrocket on high usage
Scalability	Typically requires capacity planning and manual resource allocation.	Automatically scales in response to the number of incoming requests.
Development speed	Can be slower due to the need for setting up and managing infrastructure.	Can be faster due to the abstraction of infrastructure management, allowing developers to focus on writing code.
Deployment	Deployments can be complex and time-consuming, involving multiple steps and potential downtime.	Deployments are typically simpler and faster, often involving a single command or automated process.
Performance	Performance can be consistent, but may require manual optimization and scaling to maintain.	Performance can be variable, with cold starts potentially causing latency issues.
Flexibility	Offers more control over the environment and configuration, allowing for greater customization.	Limited control over the environment, with developers relying on the cloud provider's configurations and runtimes.
Vendor lock-in	Less prone to vendor lock-in, as applications can be more easily migrated between different infrastructure providers.	More prone to vendor lock-in, as applications are often tightly coupled with the specific serverless platform and its services.

Table 9.1: *Key aspects of serverless development*

Serverless has its roots in **Platform as a Service** (**PaaS**) and is arguably a form of PaaS. Serverless is far more restrictive than a typical PaaS, and this lets the provider of the Microservice scale it aggressively and provide fault tolerance capabilities such as Kubernetes, without the configuration overhead.

One such restriction is a short timeout. This fail-fast approach ensures that serverless deployments are responsive and can scale sensibly. Since billing in serverless is tied to CPU time, it is vital to keep serverless code efficient and responsive to keep the cloud bill low.

Using AWS Lambda

AWS Lambda is a serverless computing service that allows you to run your code without provisioning or managing servers. We will walk through the

process of creating, deploying, and testing a simple Lambda function using Java. We will use the AWS Management Console, AWS CLI, and AWS SDK for Java to work with AWS Lambda.

For this, we will need an AWS account. We would normally want an **Identity Access and Management** (**IAM**) account and not the "root" account. By setting up an additional identity, we remove some of the risk of hacking. It also makes it easier for us to delegate responsibility to teammates without exposing private information such as billing. Setting up an administrative account for AWS is described as follows: **https://docs.aws.amazon.com/IAM/latest/UserGuide/getting-set-up.html**

It is important to grant proper permissions to the new account responsible for Lambda access. In *figure 9.1*, we see the creation of a new user responsible for working with Lambda. This new account is granted the **AWSLambda_FullAccess** permissions:

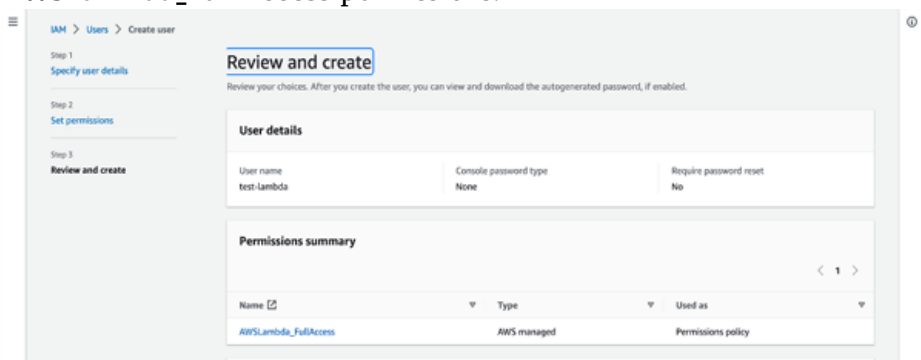

Figure 9.1: *Permissions for a new IAM administrator account*

Once such a user exists, we will need to generate an access token with which we can programmatically deploy code to Lambda. We accomplish this with Access Keys that we will use in the AWS configuration on our machine. You can read more about access keys and the way to generate them as follows: **https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html**

Notice that once access keys are generated, you cannot recover them. Make sure to write down both keys. If not, you will need to generate a completely new key. *Figure 9.2* shows the Web interface for creating a new access key:

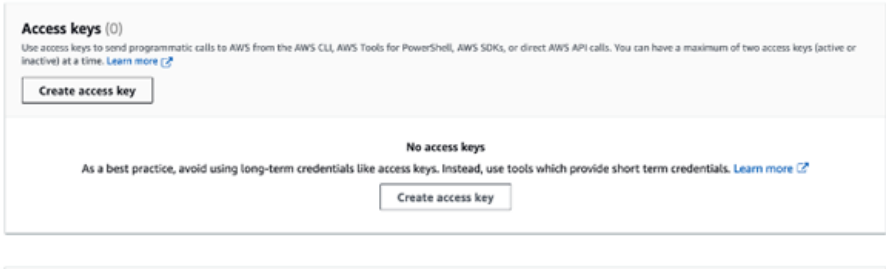
Access keys (0)
Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more
Create access key
No access keys
As a best practice, avoid using long-term credentials like access keys. Instead, use tools which provide short term credentials. Learn more
Create access key

Figure 9.2: *Access key creation UI*

To build our first Lambda, we will work with the AWS Toolkit for IntelliJ/IDEA. But before we can use that, we need to install SAM which is the AWS command line tool used internally by the AWS Toolkit. You can follow the instructions here to install SAM: **https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/install-sam-cli.html**

Once installed, we can install the AWS Toolkit from the Plugins section in the IntelliJ/IDEA Settings, as shown in *figure 9.3*:

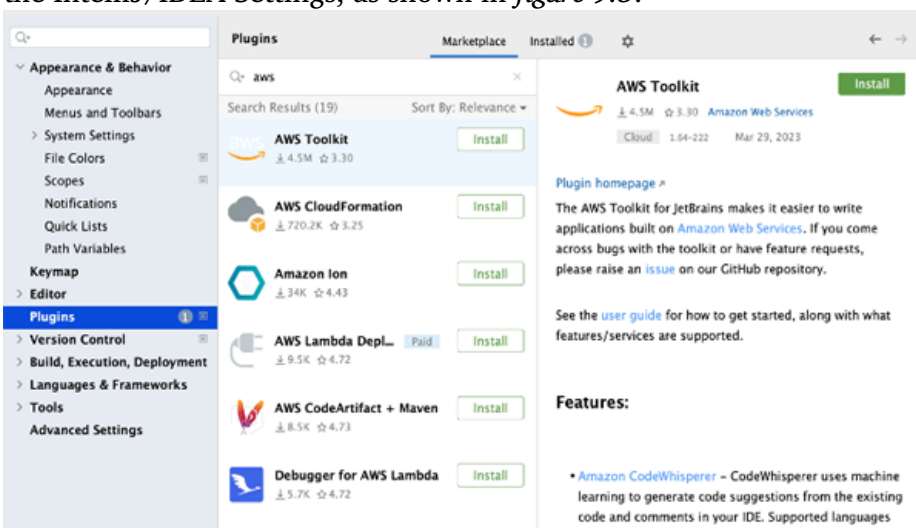

Figure 9.3: *Installing the AWS Toolkit*

The AWS Toolkit will prompt you to configure the AWS credentials upon IDE restart. At that point, you can copy the credentials from the access key creation and save. Once saved, the toolkit should be configured. In some cases, the Toolkit fails to locate the SAM installation. If this happens to you, the solution is to set it in the system settings, as shown in *figure 9.4*:

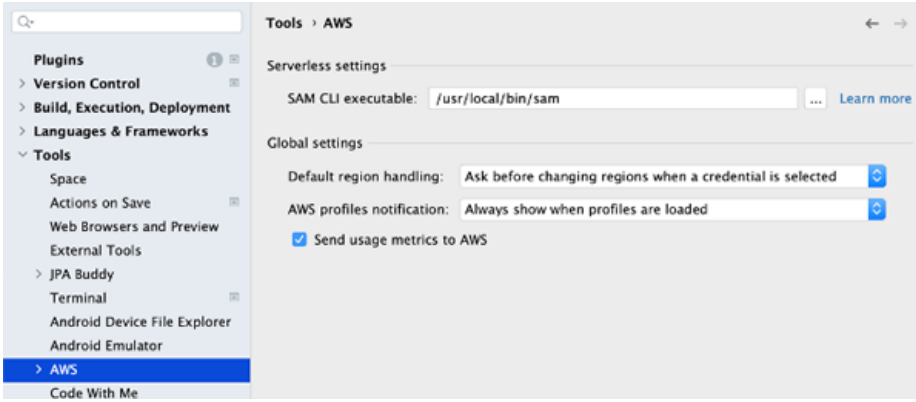

Figure 9.4: *Location of the SAM CLI*

Once this is done, we should be able to create a new AWS hello world application, as shown in *figure 9.5*. Notice that for this to work, Docker must be running and Maven needs to be in the IDE system path:

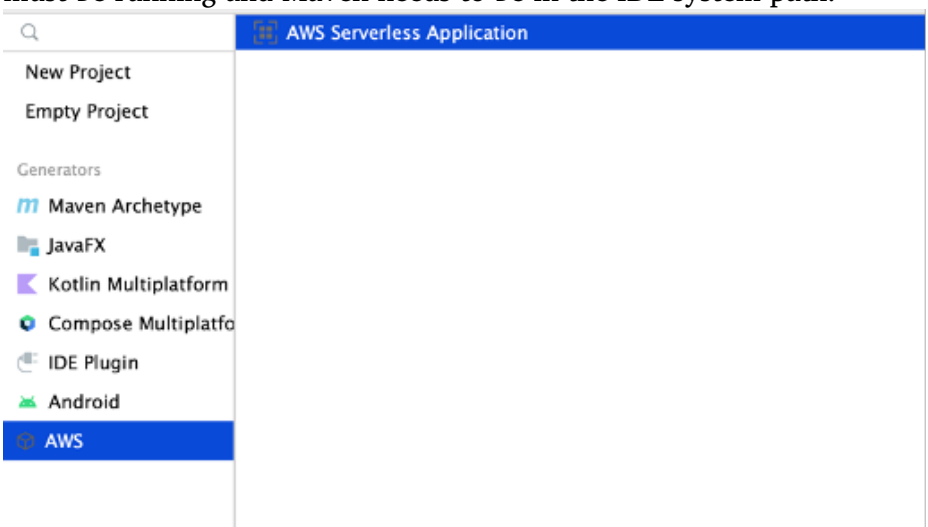

Figure 9.5: *New serverless application*

When configuring a new Lambda project, we need to make several decisions as shown in *figure 9.6*. We need to pick the version of the JDK to work with. Currently, Lambda only supports JDK 8 and JDK 11, out of the box. We can deploy more arbitrary containers with newer versions of the JDK but that is outside of the scope of this book. There are plans to support JDK 17 and probably 21 as both are **Long Term Support** (**LTS**) releases:

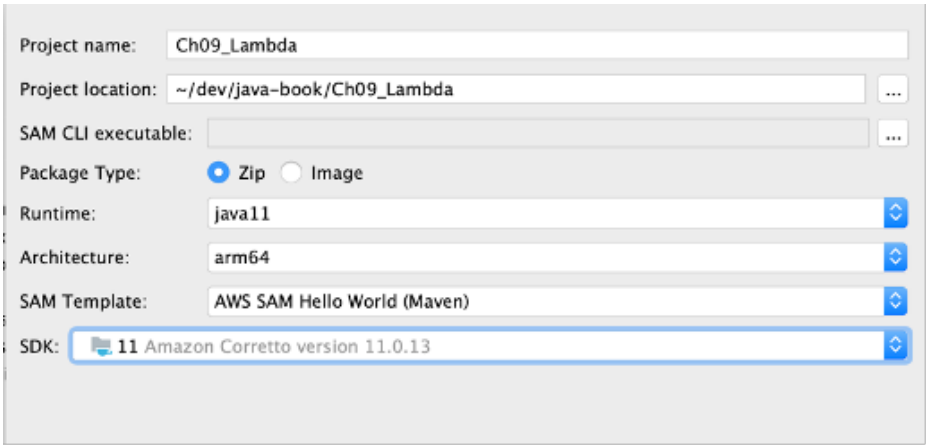

Figure 9.6: *New serverless project creation*

Other than the Java version, we can also decide if we wish to use Arm64 or ×86 architecture. One of the advantages of working with Java is the relative seamlessness in transitioning between the two. Arm64 is noticeably cheaper than ×86; this can be important during deployment.

When running for the first time, we get prompted with a dialog to set up our local run environment, as shown in *figure 9.7*. We can configure all the settings in the dialog. Notice that the input can be set to a file already present under the events directory:

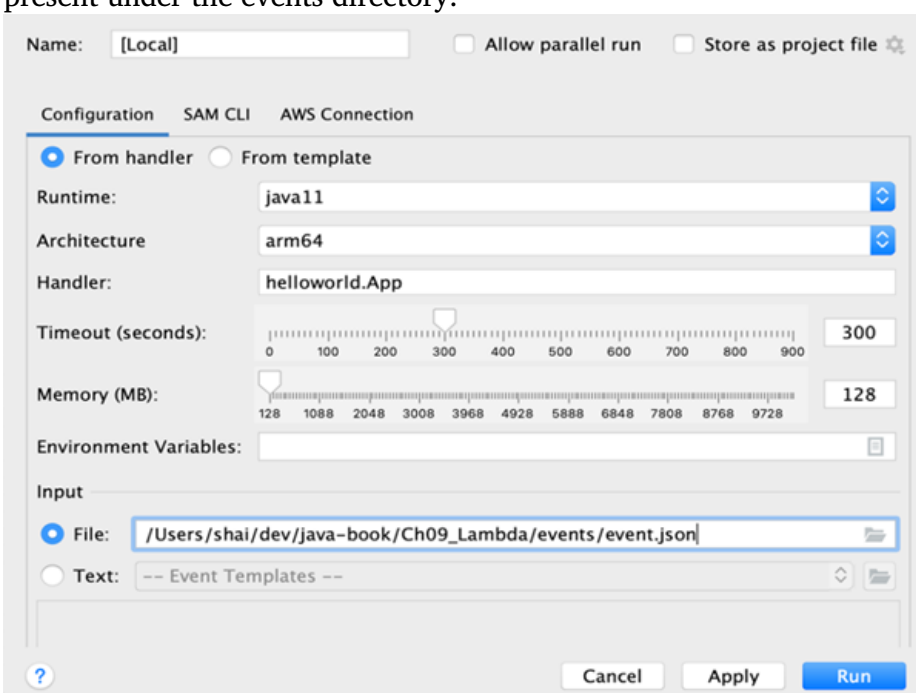

Figure 9.7: *Local execution run dialog*

The following code for Hello World is relatively simple:

```
1. public class App implements RequestHandler<APIGatewayProxyRe-
   questEvent, APIGatewayProxyResponseEvent> {
2.
3.     public APIGatewayProxyResponseEvent handleRequest(final
       APIGatewayProxyRequestEvent input, final Context context) {
4.         Map<String, String> headers = new HashMap<>();
5.         headers.put("Content-Type", "application/json");
6.         headers.put("X-Custom-Header", "application/json");
7.
8.         APIGatewayProxyResponseEvent response = new
           APIGatewayProxyResponseEvent()
9.                 .withHeaders(headers);
10.        try {
11.            final String pageContents = this.
               getPageContents("https://checkip.amazonaws.com");
```

```
12.             String output = String.format("{ \"message\": \"hello
                world\", \"location\": \"%s\" }", pageContents);
13.
14.             return response
15.                     .withStatusCode(200)
16.                     .withBody(output);
17.         } catch (IOException e) {
18.             return response
19.                     .withBody("{}")
20.                     .withStatusCode(500);
21.         }
22.     }
23.
24.     private String getPageContents(String address) throws
        IOException{
25.         URL url = new URL(address);
26.         try(BufferedReader br = new BufferedReader(new
            InputStreamReader(url.openStream()))) {
27.             return br.lines().collect(Collectors.joining(System.
                lineSeparator()));
28.         }
29.     }
30. }
```

A typical Lambda function implements the **RequestHandler** interface. It handles an incoming request and returns the page contents from the AWS checkIP website as a JSON response. This might not seem very useful, but this is an ideal Microservice and that is at the core of serverless architecture. Serverless is the ultimate Microservice and enforces the principles behind Microservices religiously.

Much like other hello world implementations, this code is far from the "real world". It has no storage or database access. This is where serverless become "tricky".

GraalVM and Monolith serverless

Lambda supports building a function using GraalVM, which significantly improves startup time. That is incredibly important for Lambda, as it lets AWS spin up additional instances fast. It means you can configure the number of warm instances to a small number. The reduction in RAM is also helpful, as we can keep the serverless requirements minimal. Both can

significantly reduce serverless bills.

Unfortunately, GraalVM support is missing from the current version of the plugin. For this, we will need to use the SAM command line tool. We create a new directory and invoke:

sam init

This prompts us with the following options:

Which template source would you like to use?

1 - AWS Quick Start Templates

2 - Custom Template Location

Choice: 1

We pick Option 1 for the quick start template. This leads to the following options:

Choose an AWS Quick Start application template

1 - Hello World Example

2 - Multi-step workflow

3 - Serverless API

4 - Scheduled task

5 - Standalone function

6 - Data processing

7 - Hello World Example With Powertools

8 - Infrastructure event management

9 - Serverless Connector Hello World Example

10 - Multi-step workflow with Connectors

11 - Lambda EFS example

12 - DynamoDB Example

13 - Machine Learning

Template: 1

Again, we pick Option 1. You can pick a different option from the list, but GraalVM might not be available in every option. This produces the following options:

Which runtime would you like to use?

1 - aot.dotnet7 (provided.al2)

2 - dotnet6

3 - dotnet5.0

4 - dotnetcore3.1

5 - go1.x

6 - go (provided.al2)

7 - graalvm.java11 (provided.al2)

8 - graalvm.java17 (provided.al2)

9 - java11

10 - java8.al2

11 - java8

12 - nodejs18.x

13 - nodejs16.x

14 - nodejs14.x

15 - nodejs12.x

16 - python3.9

17 - python3.8

18 - python3.7

19 - python3.10

20 - ruby2.7

21 - rust (provided.al2)

Runtime: 8

Here, we can select Option 8, GraalVM with Java 17. This will leave a few options:

Based on your selections, the only Package type available is Zip.

We will proceed to selecting the Package type as Zip.

```
Which dependency manager would you like to use?

    1 - gradle

    2 - maven

Dependency manager: 2


Would you like to enable X-Ray tracing on the function(s) in your
application? [y/N]:


Would you like to enable monitoring using CloudWatch Application Insights?

For more info, please view https://docs.aws.amazon.com/AmazonCloudWatch/
latest/monitoring/cloudwatch-application-insights.html [y/N]:


Project name [sam-app]: Ch09_Graal_Lambda


    -----------------------

    Generating application:

    -----------------------

    Name: Ch09_Graal_Lambda

    Runtime: graalvm.java17 (provided.al2)

    Architectures: x86_64

    Dependency Manager: maven
    Application Template: hello-world

    Output Directory: .

    Configuration file: Ch09_Graal_Lambda/samconfig.toml


    Next steps can be found in the README file at Ch09_Graal_Lambda/
    README.md



Commands you can use next

=========================

[*] Create pipeline: cd Ch09_Graal_Lambda && sam pipeline init --bootstrap

[*] Validate SAM template: cd Ch09_Graal_Lambda && sam validate

[*] Test Function in the Cloud: cd Ch09_Graal_Lambda && sam sync --stack-
name {stack-name} --watch
```

You can use the defaults for most of the rest, but in the first option, Maven

was selected instead of Gradle. With this, you will have a project you can open in the IDE and run. The code itself is identical to the previous Lambda, but it is designed to work with GraalVM running on Java 17. The readme file in the root of the project contains the build instructions to generate the native Docker image that we can use to submit the application to Lambda.

Since Docker is used for Lambda submission, we can package applications that are far more sophisticated. In fact, we can package a Spring Boot application as a Lambda function and gain the benefits of seamless management. We can package a Monolithic Spring Boot application as serverless, if we handle state properly and if multiple app instances will work correctly. This is a surprisingly common use case for Lambda; it negates some of its key benefits, but it still works reasonably well.

The cloud ecosystem

Serverless is often used as cloud duct tape. That is not a disparaging remark; duct tape is fantastic. It can intercept a Web hook at scale and low cost, then redirect the output of the Web hook to the location we need. It can serve as a connector between disparate systems without the complexity of provisioning servers, administration, and scaling.

Eventually, operations in an application need to write something down somewhere. That is where serverless applications become tricky. Serverless is priced very competitively, for example, AWS provides a million free requests per month[1]. But as most would say, the devil is in the details. When we need to write something down, we need to pay extra. This is true when working in a container environment too, as writing over there is problematic as the container could be destroyed. We need to write to a volume that would remain. In a typical application, we would set up a database that would write to an external volume. This volume will remain even if we restart the database container.

With serverless, we need to store the information somewhere, and this leaves us with a few options:

- Invoke another service.
- Use a Database or storage API.

If one of these is hosted outside of AWS, we will be charged for egress/ingress fees, which are charged for traffic going in and out of AWS. If these services are inside of AWS such as S3 object storage, you would be charged for that. The advantage serverless has is in using the entire ecosystem. Since it is hosted in the same datacenter as the cloud services, the performance of using said additional services is great. However, it means

that the low cost of serverless is misleading. To truly evaluate the cost of serverless, we need to include the additional services. If you already make use of such services, you might end up saving.

Conclusion

Serverless is the natural extension of Microservices. *Figure 9.8* shows how serverless fits in the grand scheme of Monolith and serverless. Ultimately, we trade some control for ease of development and a reduction in DevOps costs. Please refer to the following figure:

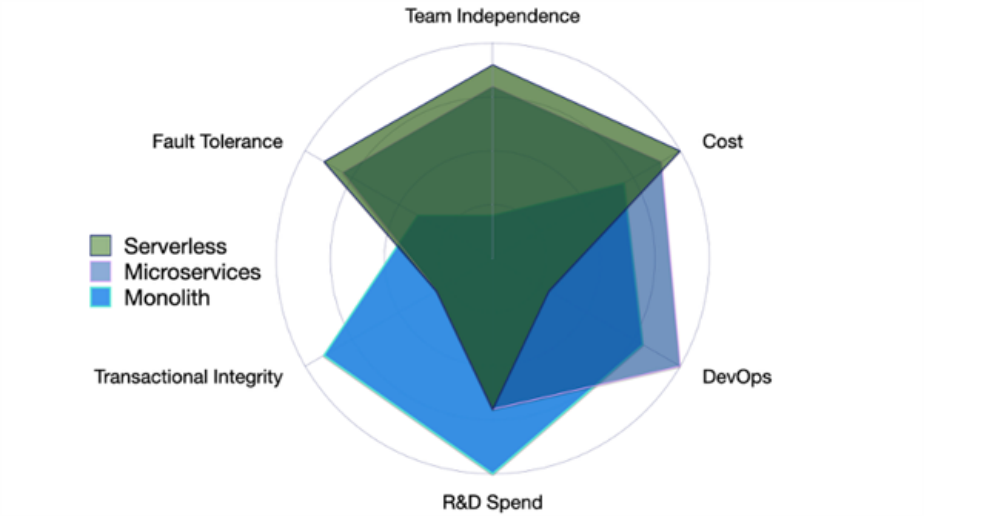

Figure 9.8: *Radar chart contrasting serverless, Microservices, and Monolith*

Points to remember

- Serverless lets us focus on coding and reduces the hassle of DevOps-related tasks.
- The tradeoff is costly; serverless can be very cheap and then spike.
- Serverless makes sense as a part of a cloud infrastructure. Costs and vendor lock-in on the cloud should be taken into consideration.
- We can deploy a Monolith as a serverless function.
- GraalVM can be used to reduce costs by reducing startup time and instances.

Multiple choice questions

1. The AWS tool for serverless development is called:

a. SAM

b. AWS

c. ATool

d. IAM

e. ADEP

2. GraalVM is:

a. The default Lambda deployment target

b. Natively supported by Lambda

c. Can run via Docker on Lambda

d. Is coming in an upcoming update to Lambda

3. On Lambda we can deploy:

a. A single Web service

b. Multiple services

c. A docker container

d. A Monolith

e. All the above

Answers

1. a

2. c

3. e

---

1 https://aws.amazon.com/lambda/pricing/

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 10
Monitoring and Observability

Introduction

Monitoring and observability are essential aspects of software development, which help developers and operations teams ensure the reliability, stability, and performance of their applications. As systems become increasingly complex, the need for effective monitoring and observability tools grows in tandem, providing a comprehensive understanding of the behavior and health of applications during development, deployment, and production. The Java ecosystem is no exception, offering a rich set of libraries and tools designed to facilitate these critical tasks. This chapter will delve into the concepts of monitoring and observability in the context of Java applications, exploring the underlying principles, methodologies, and best practices that will enable developers to build robust, scalable, and resilient systems.

Java developers have at their disposal, a wide range of monitoring and observability tools, ranging from built-in JVM features to third-party libraries and frameworks. These tools empower developers to collect, analyze, and visualize various types of metrics, logs, and traces to diagnose performance bottlenecks, detect anomalies, and troubleshoot issues. Additionally, they aid in capacity planning, resource management, and the optimization of infrastructure components. In this chapter, we will examine some of the most popular and effective monitoring and observability tools available for Java, discussing their strengths and weaknesses, as well as providing practical examples and use cases to help you select the right tools for your specific application requirements.

Structure

In this chapter, we will discuss the following topics:

- What is monitoring?

- Pillars of observability
- Prometheus
- Grafana
- Micrometer
- Actuator
- Developer observability

Objectives

In this chapter, we will develop an insight into the production state of the application. We will be able to understand how to better facilitate that state and how to use it to improve the quality of our application.

What is monitoring?

Monitoring refers to the systematic process of gathering, processing, and analyzing data from various sources within an application, its infrastructure, and its runtime environment. This data can include metrics, logs, and traces that provide insight into the performance, availability, and functionality of the system. The primary objective of monitoring is to maintain the overall health of the application and ensure it meets the desired level of service and performance for end-users.

Monitoring encompasses several key aspects, such as collecting performance data, setting up alerts and notifications, establishing baseline performance expectations, and identifying trends or anomalies. By doing so, developers and operations teams can proactively detect and resolve issues before they escalate into critical incidents that could result in system downtime, degraded performance, or even data loss. Furthermore, monitoring enables teams to continuously improve their applications by analyzing the collected data, identifying bottlenecks, optimizing resource usage, and refining their infrastructure to better support the applications evolving needs.

An effective monitoring strategy should be comprehensive, covering various aspects of the application, including its infrastructure, runtime environment, and code execution. This can be achieved by using a combination of tools and techniques, such as log analyzers, performance profilers, and **Application Performance Management** (**APM**) solutions. In the world of Java, there are many monitoring tools available, both built-in and third-party, that cater to the unique characteristics of the Java runtime environment and the specific requirements of Java-based applications. By leveraging these tools and adopting a robust monitoring strategy, developers can build and maintain high-quality, resilient, and scalable applications that meet the needs and expectations of their users.

Figure 10.1 shows the typical layers in a monitored Java application:

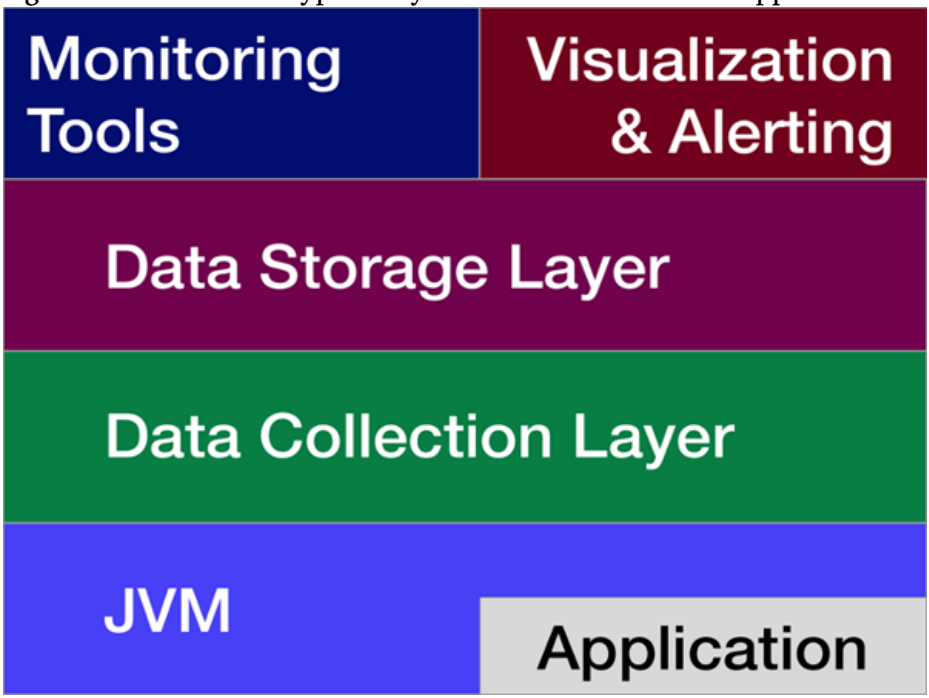

Figure 10.1: *Layers of a monitored application*

The Application Layer represents the Java application itself, including its code, libraries, and frameworks. It is running within the JVM which provides a great deal of the core infrastructure for the data collection layer.

The Data Collection Layer is responsible for collecting metrics, logs, and traces from the application, runtime environment, and infrastructure. There are different data sources, such as log files, JVM metrics (for example, CPU usage, memory allocation, and garbage collection), and custom instrumentation.

One of the biggest parts of a monitored application is Data Storage. This seems like a simple enough problem until we take the scale and frequency into account. Every request to the servers can result in many datapoints to keep track of. These add up quickly and we need fast access to that data to provide visualizations.

The Monitoring tools include built-in and third-party monitoring tools that are used to gather, process, and analyze the collected data. Examples include JMX, Java Flight Recorder, Logstash, and APM solutions like New Relic or Dynatrace.

Visualization and Alerting include dashboards, charts, and graphs used to

visualize the collected data, as well as the alerting mechanisms in place for notifying developers and operations teams about potential issues.

One thing that is hard to represent in such a diagram is the Feedback Loop. It links the visualization and alerting stage back to the application layer due to the ongoing nature of monitoring and its function in enhancing application performance and dependability. As we gain insights from monitoring, we use them to make informed decisions and adjustments. This, in turn, translates to changes in the visualizations that we can use to verify the impact of our changes.

Pillars of observability

Observability is a term that has gained significant traction in recent years within the software development community. While it shares some similarities with monitoring, it represents a broader and more holistic approach to understanding the internal state of a system. Observability emphasizes the ability to ask arbitrary questions about an application's behavior and performance, especially in complex and distributed systems where failures and issues can be challenging to predict and diagnose.

In contrast to monitoring, which often focuses on predefined metrics and alerts, observability is about providing developers and operations teams with a detailed and dynamic view of the system's internal workings. This allows them to explore, experiment, and learn from the data collected, enabling them to proactively identify and resolve issues, optimize performance, and make data-driven decisions.

The three pillars of observability

Observability is often described in terms of three main pillars, which are metrics, logs, and traces. These pillars represent different types of data that, when combined, offer a comprehensive understanding of an application's behavior and performance.

1. **Metrics:** Metrics are numerical representations of data collected over time, often in the form of time-series data. Examples of metrics include request rates, error rates, latency, CPU usage, and memory consumption. Metrics provide a high-level overview of the system's performance and health, allowing teams to track and compare trends over time and identify anomalies or potential issues.
2. **Logs:** Logs are records of events that occur within an application or its infrastructure. They can include information such as system messages, error messages, warnings, or even custom events added by developers. Logs offer a more granular view of the system, enabling developers to pinpoint specific issues or occurrences and gain insights

into the sequence of events that led to a particular outcome.

3. **Traces:** Traces are particularly relevant in distributed systems, where a single request may span multiple services or components. Tracing involves tracking the flow of a request through the system and recording information about each component it interacts with along the way. Traces provide a detailed view of the request's journey, making it easier to identify bottlenecks, latency issues, and other performance-related concerns in complex, multi-service architectures. *Figure 10.2* shows a tracing view from **uptrace.dev**, where we can see individual operations related to a single request broken down into their applicable components:

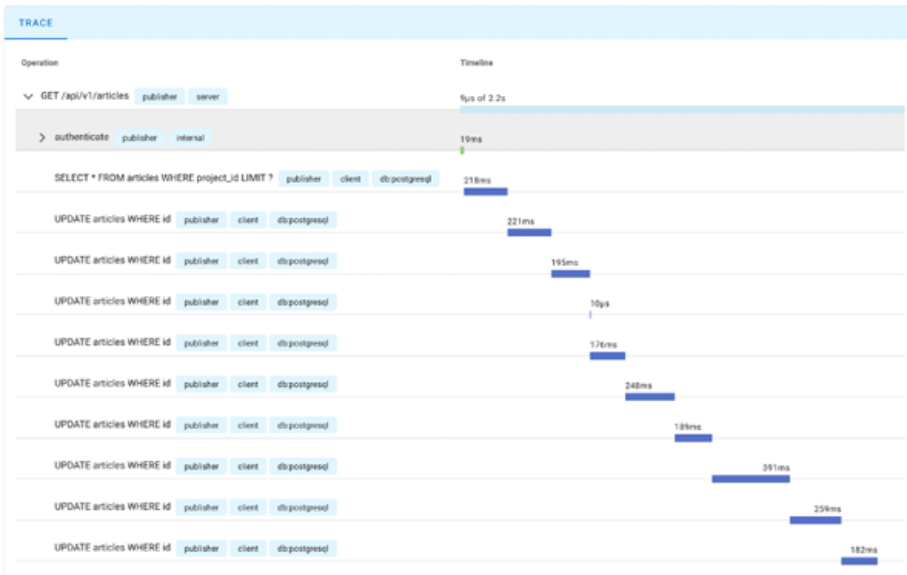

Figure 10.2: *Tracing span view in uptrace.dev*

By incorporating all three pillars of observability into their development and operations practices, teams can build a more in-depth understanding of their applications, leading to more resilient, performant, and reliable systems. In the context of Java applications, numerous tools and libraries are available to help developers achieve observability, including built-in JVM features, third-party libraries, and external services, that focus on metrics, logs, and traces. Embracing observability as a core principle of software development is crucial for teams looking to build and maintain high-quality Java applications in today's rapidly evolving technology landscape.

What makes a system observable?

An observable system goes beyond merely collecting and presenting data;

it empowers developers and operations teams to ask meaningful questions about the application's behavior and performance and derive actionable insights from the collected data. This requires not only capturing the right data from various sources but also structuring and analyzing it in a way that enables teams to explore, experiment, and learn. A truly observable system allows for deep introspection and fosters a culture of continuous improvement and data-driven decision-making.

To achieve an observable system, the collected data from metrics, logs, and traces must be organized, stored, and indexed in a manner that facilitates efficient querying and analysis. This may involve using specialized data stores, such as time-series databases for metrics or log management solutions for logs, and adopting standardized data formats to ensure seamless integration across various tools and services. Additionally, visualization and analysis tools should be used to make the data more accessible and understandable, allowing teams to ask complex questions and receive informed answers quickly.

Having an observable system enables teams to:

- **Proactively identify and resolve issues:** By continuously monitoring the system's behavior and performance, teams can detect and address potential problems before they escalate into critical incidents, ensuring the application remains performant and available to end-users.
- **Perform root cause analysis:** When issues do occur, an observable system allows teams to quickly pinpoint the root cause by examining the collected data, tracing requests, and analyzing logs, minimizing the time spent on troubleshooting and reducing the impact on users.
- **Optimize performance and resource usage:** By examining the system's metrics and traces, teams can identify performance bottlenecks, optimize resource allocation, and improve the overall efficiency of their application and infrastructure.
- **Make informed decisions:** Observable systems provide valuable insights that can inform decisions about capacity planning, infrastructure changes, and feature development, ensuring that teams can adapt and evolve their applications to meet the changing needs and expectations of their users.

The key issue is our ability to ask questions and provide an informed, fact-based answer. What is the average level of load on our servers?

Can we cut down the number of container instances or do we need to increase it?

Why was there a slowdown on Monday at 8 p.m.?

In a properly observable system, we can answer these questions, or at least provide a path to answer these questions.

Prometheus

Prometheus is an open-source monitoring and alerting system that has become increasingly popular in the world of software development, particularly for cloud-native and containerized applications. It was initially developed by SoundCloud in 2012 and has since been adopted as a project under the **Cloud Native Computing Foundation** (**CNCF**). Prometheus is designed to handle the complex monitoring requirements of modern, distributed systems by providing a robust, scalable, and extensible solution for gathering, storing, and analyzing metrics.

The key features of Prometheus include the following:

- **Multi-dimensional data model:** Prometheus uses a powerful data model based on key-value pairs called labels, which allow developers to define and query metrics with greater flexibility and granularity. This makes it easy to represent and organize data from various sources, such as microservices, containers, and cloud infrastructure components.
- **Powerful query language:** Prometheus features a versatile query language called **Prometheus Query Language (PromQL**), which enables developers and operations teams to perform complex queries on the collected data. With PromQL, users can aggregate, filter, and transform metrics to gain deeper insights into their system's behavior and performance.
- **Pull-based data collection:** Unlike many monitoring solutions that rely on pushing data to a central server, Prometheus follows a pull-based approach, where it actively scrapes metrics from target systems at regular intervals. This reduces the risk of data loss and makes it easier to monitor dynamic environments, such as container orchestration platforms like Kubernetes.
- **Built-in alerting:** Prometheus includes a built-in alerting system that allows teams to define and manage alerts based on specific conditions or thresholds. When an alert is triggered, Prometheus can send notifications through various channels, such as e-mail, Slack, or PagerDuty, ensuring that teams can respond quickly to potential issues.
- **Integration with Grafana:** Prometheus can be easily integrated with Grafana, a popular open-source visualization platform, to create interactive and customizable dashboards. This allows teams

to visualize the collected metrics and gain a better understanding of their system's health and performance.

Prometheus provides client libraries for various programming languages, including Java. The Java client library allows developers to instrument their Java applications by exposing custom metrics that can be scraped by a Prometheus server. Additionally, the library includes built-in support for monitoring **Java Virtual Machine** (**JVM**) metrics, such as memory usage, garbage collection, and thread counts, providing valuable insights into the runtime environment. *Figure 10.3* shows Prometheus in action:

Figure 10.3: *Prometheus in action*

Spring Boot 3.x standardized on a micrometer, which we will discuss later in the chapter. It is the best way to integrate Prometheus into your application.

Grafana

Grafana is an open-source platform for analytics and visualization that has gained widespread popularity in the DevOps and monitoring communities. Grafana enables users to create dynamic and interactive dashboards to visualize time-series data, logs, and traces from various data sources, including Prometheus, InfluxDB, Elasticsearch, and many others. By providing a comprehensive and user-friendly interface, Grafana helps developers and operations teams gain valuable insights into their application's behavior and performance, enabling them to make data-driven decisions and improve system reliability.

The key features of Grafana include the following:

- **Data source agnostic:** Grafana supports a wide range of data

sources, allowing users to visualize and analyze data from different monitoring systems, databases, and services. This flexibility makes Grafana an ideal choice for organizations with diverse technology stacks or those looking to migrate between monitoring solutions.

- **Rich visualization options:** Grafana offers a variety of visualization options, including graphs, tables, heatmaps, and gauges, allowing users to represent data in a way that best suits their needs. Custom plugins can also be developed to create new visualization types or extend existing ones.
- **Customizable dashboards:** Grafana's dashboards are highly customizable, enabling users to create layouts that meet their specific requirements. Users can create multiple panels with different visualization types, apply filters and variables, and organize them into rows and columns for easy navigation.
- **Alerting:** Grafana includes a built-in alerting system that allows users to define alerts based on specific conditions or thresholds. Alerts can be configured to send notifications through various channels, such as e-mail, Slack, PagerDuty, or Webhook, ensuring that teams can respond quickly to potential issues.
- **Collaboration and sharing:** Grafana enables users to share dashboards and panels with other team members or stakeholders, promoting collaboration and knowledge sharing. Dashboards can be exported as JSON files, embedded in Web pages, or shared through a direct link.

Grafana integrates well with Prometheus and provides a sophisticated user interface that goes far beyond the limits of Prometheus, as can be seen in *figure 10.4*:

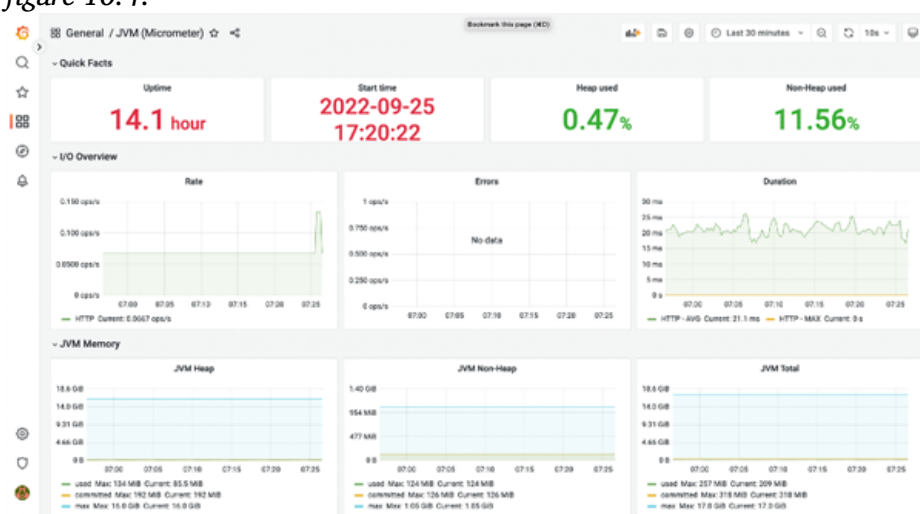

Figure 10.4: *Grafana in action*

Micrometer

Micrometer is a metrics instrumentation library for Java applications, designed to serve as a facade between your application code and various monitoring systems. It provides a vendor–neutral interface that allows developers to instrument their applications with minimal effort and without being locked into a specific monitoring solution. Micrometer has become the go-to metrics library for Spring Boot applications, starting with version 2.0, and it is an essential tool for building observable Java applications.

The key features of a micrometer include the following:

- **Vendor–neutral API:** Micrometer offers a simple and consistent API for instrumenting Java applications, regardless of the underlying monitoring system. This enables developers to write code that can be easily integrated with multiple monitoring solutions, increasing flexibility and reducing vendor lock-in.
- **Rich set of metrics types:** Micrometer provides various metric types, such as counters, gauges, timers, and distribution summaries, allowing developers to collect a wide range of metrics relevant to their application's behavior and performance.
- **Integration with monitoring systems:** Micrometer includes built-in support for several popular monitoring systems, such as Prometheus, InfluxDB, Datadog, and Graphite. Additionally, it allows developers to implement custom registries for other monitoring systems.
- **Hierarchical and dimensional metrics:** Micrometer supports both hierarchical and dimensional data models, enabling developers to choose the best approach for their application and monitoring system.

Observability in Spring Boot 3.x is standardized on a micrometer. To take advantage of this, we need to use the **ObservationRegistry** to register an **ObservationHandler**. This handler can produce observability events as such:

1. ObservationRegistry registry = ObservationRegistry.create();
2. registry.observationConfig()
3. .observationHandler(new ObserveHandler());
4.
5. Observation.createNotStarted(name, registry)
6. .observe(() -> ...);

In the observe method, we can log whenever an observable event occurs or write the data to a database. However, the real value of the micrometer comes from its usage with the Spring actuator.

Actuator

Spring Boot actuator is a module within the Spring Boot framework that provides built-in production-ready features for monitoring and managing Spring Boot applications. The actuator exposes various endpoints that offer valuable insights into the application's health, metrics, and configuration properties. This enables developers and operations teams to efficiently monitor the application's behavior, diagnose issues, and fine-tune performance.

The key features of the Spring Boot actuator include the following:

- **Health indicators:** Actuator includes a built-in health endpoint that provides a high-level overview of the application's health. It aggregates information from various health indicators, such as database connectivity, disk space, and external service availability, to give a comprehensive view of the application's status.
- **Metrics:** The actuator integrates with the Micrometer library to collect and expose various application metrics, including JVM, CPU, and memory usage, as well as custom application-specific metrics. These metrics can be exported to external monitoring systems, such as Prometheus, InfluxDB, or Graphite, for further analysis and visualization.
- **Configuration properties:** Actuator exposes an endpoint to view the application's configuration properties, including those from external configuration sources, such as environment variables or configuration files. This helps in validating and troubleshooting configuration-related issues.
- **Auditing:** Actuator provides an auditing feature that automatically records significant events, such as user authentication, and exposes them through an endpoint for analysis and auditing purposes.
- **Application information:** Actuator exposes endpoints to display information about the application, such as the build version, commit information, and deployed environment details. This information is useful for troubleshooting and identifying the application version in production.
- **Custom endpoints:** Actuator allows developers to create custom endpoints for application-specific monitoring and management tasks. This enables the addition of custom application logic to the built-in actuator functionality.

Enabling and configuring Spring Boot actuator

To enable Spring Boot actuator in your Spring Boot application, add the spring-boot-starter-actuator dependency to your build configuration by changing the **pom.xml** file as follows:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-actuator</artifactId>
4. </dependency>

By default, only the health and info endpoints are exposed over HTTP. To expose additional endpoints or configure their behavior, modify the **application.properties** file as follows:

1. management.endpoints.web.exposure.include=health,info,metrics
2. management.endpoint.health.show-details=always

The JHipster application we used in *Chapter 6, Testing and CI*, demonstrates the observability capabilities of Spring Boot including a custom administration UI, as shown in *figure 10.5*:

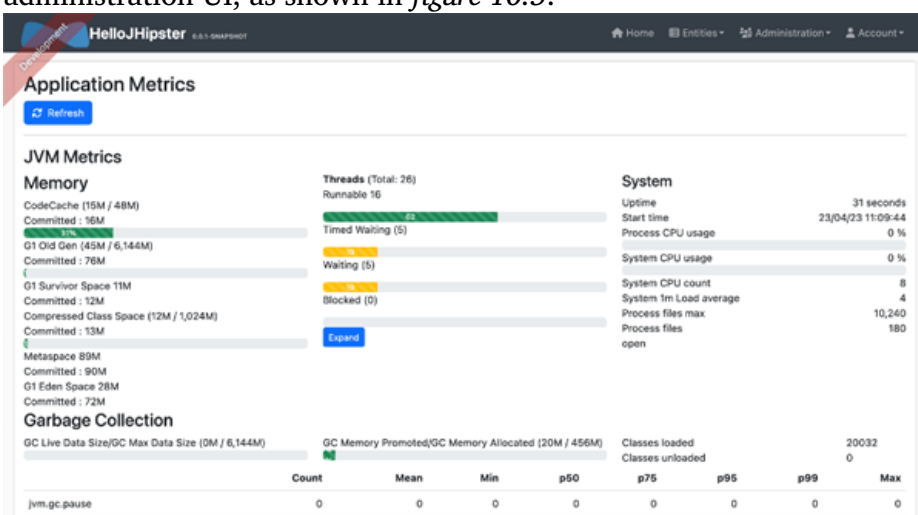

Figure 10.5: *Custom administration UI in JHipster*

Once enabled, we can use both **Java Management Extensions** (**JMX**) tools as well as actuator-dedicated tools to observe a Spring Boot application. In *figure 10.6*, we can see the native support for the Spring Boot actuator built into IntelliJ/IDEA Ultimate edition:

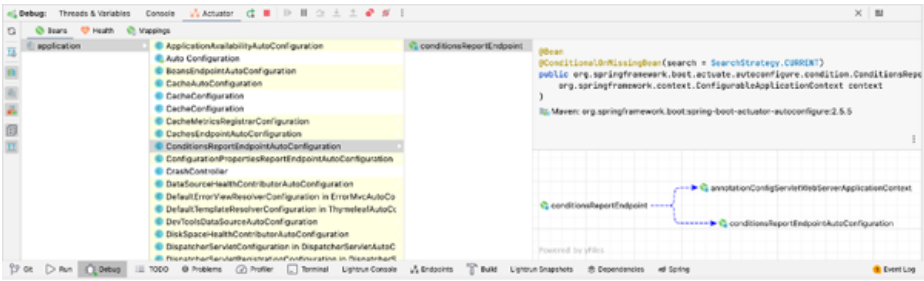

Figure 10.6: *Actuator management from IntelliJ/IDEA Ultimate edition*

Exposing custom information via Spring Boot actuator

Spring Boot actuator provides an easy way to expose custom information about your application through the info endpoint. This can include details such as application-specific properties, build versions, or even dynamic runtime information. Exposing custom information via the Actuator can help you quickly diagnose issues, track application versions in production, and provide better insights into your application's behavior.

We can get started by implementing a custom **InfoContributor** bean that adds custom data to the **Info.Builder**. We begin by creating a class that implements the **InfoContributor** interface:

```
@Component
public class CustomInfoContributor implements InfoContributor {

@Override
public void contribute(Info.Builder builder) {
builder.withDetail("custom.runtimeInfo", getRuntimeInfo());
}

private Map<String, Object> getRuntimeInfo() {
Map<String, Object> runtimeInfo = new HashMap<>();
runtimeInfo.put("currentTime", LocalDateTime.now());
runtimeInfo.put("activeThreads", Thread.activeCount());
// Add more custom dynamic information as needed
return runtimeInfo;
}
}
```

We then register the **CustomInfoContributor** bean in our Spring Boot application:

```
1. @Configuration
2. public class CustomInfoConfiguration {
3.
4. @Bean
5. public CustomInfoContributor customInfoContributor() {
6. return new CustomInfoContributor();
7. }
8. }
```

Now, when we access **http://localhost:8080/actuator/info**, we can see the JSON contains the additional information we added:

```
1. {
2. "custom": {
3. "runtimeInfo": {
4. "currentTime": "2023-04-22T12:34:56.789",
5. "activeThreads": 42
6. }
7. }
8. }
```

Developer observability

Debugging issues on a production server might tempt developers to simply open a debug port. However, this approach is fraught with multiple challenges given as follows:

- **Restarting process:** Enabling debugging typically requires restarting the process, which may not be feasible in a production environment.
- **Disruptive breakpoints:** Breakpoints can halt execution and create problems in a production setting.
- **Security concerns:** Remote debugging protocols such as JDWP are notably insecure, with risks ranging from remote code execution to unauthorized access to source files.
- **Stability issues:** Minor errors such as poorly constructed expressions or watch variables can easily trigger a crash in the remote process.
- **Privacy risks:** Malicious developers within an organization could

exploit breakpoints to steal user credentials or escalate their privileges. This lack of oversight accounts for 60% of security breaches and may violate regulations or laws across various industries.

- **Scalability:** In a clustered environment with multiple machines, determining which machine to debug can be challenging, as any request might be directed to any machine.

Clearly, this approach is unsuitable. A more viable solution is needed to address these concerns while still providing the immense value of identifying bugs on a remote machine. Developer observability can offer such a solution by enabling real-time monitoring and insights into an application's behavior without exposing it to the aforementioned risks.

Observability enables monitoring of the production environment without deploying new code and has long been a staple of the DevOps toolchain. In today's world, managing a cluster without observability is virtually unattainable.

Developer observability adapts these principles and moves them leftward, shifting the focus from the DevOps team to the R&D team. This approach provides observability at the programming language level, directly at the "line of code" level, creating an experience akin to using a debugger instance in IntelliJ/IDEA or Visual Studio Code. However, it remains a safe observability tool that does not jeopardize the production environment's integrity.

Distinct from DevOps-oriented observability, these tools are designed to integrate seamlessly with standard development tools, such as IDEs, and work directly with source code rather than HTTP endpoints.

Injecting logs

One of the most basic features offered by most developer observability tools is log injection. This allows you to insert a new log entry into a running process directly from an IDE context menu.

You have the option to route the log to the IDE's debug console, or you can maintain the default setting, which logs it as if it were hard-coded into your application. As a result, the log entry will appear alongside all your other logs, seamlessly integrating with existing observability tools such as Elastic.

The injected log will be processed like any other log entry, appearing in the correct chronological order.

Additionally, developer observability tools enable you to print variables,

method return values, and more. Expression evaluation is performed in your chosen programming language, such as Java for JVM applications, allowing you to display variable values as needed.

The following two crucial concerns arise when using developer observability tools:

1. **Mutability:** What if a method invocation alters the application state?
2. **Performance:** What if this log significantly impacts performance?

The response to these concerns varies, depending on the tool in use. Some tools execute these expressions within a secure sandbox. Each expression is scrutinized to ensure it is read-only and does not affect the application's execution. If there is any doubt, an error will be generated, and the action will not be carried out.

This sandbox also monitors performance. If a predefined limit of CPU usage is surpassed, the action will be suspended until further clarification.

By using this approach, the remote process server code's integrity remains intact and protected from potential harm. *Figure 10.7* shows logs added in the Lightrun developer observability platform[1]:

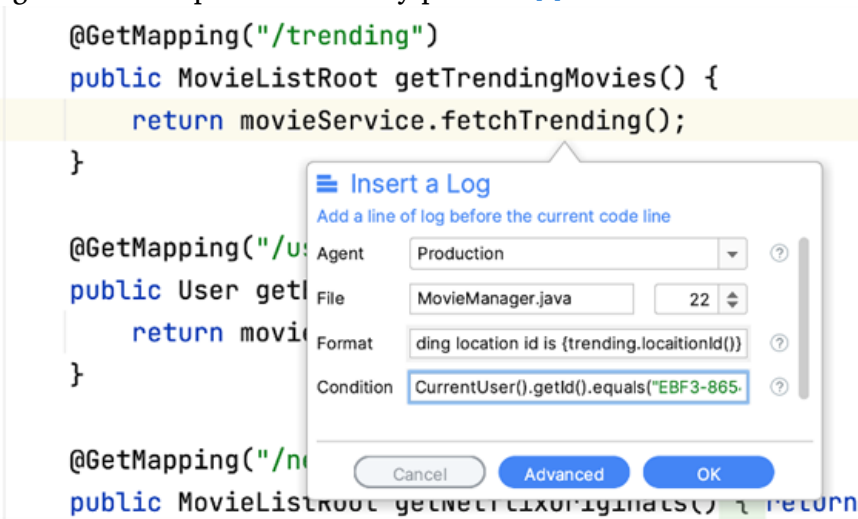

Figure 10.7: *Injecting a log using the Lightrun Developer Observability Platform*

Snapshots/captures and non-breaking breakpoints

Traditional breakpoints pose significant challenges when debugging production environments due to their disruptive nature. Snapshots, also known as Captures or Non-Breaking Breakpoints, address this issue by avoiding any interruption to the application flow.

Snapshots furnish developers with valuable information akin to what is typically provided by breakpoints in source code. This includes the ability to analyze stack traces and variable values at a specific point in time. Furthermore, snapshots support conditional expressions, similar to conditional breakpoints, allowing developers to capture information for a particular use case.

Snapshots serve as the foundation for remote debugging tools, granting unparalleled insight into production environments without sacrificing the integrity of the remote machine. By using snapshots, developers can obtain the level of understanding typically reserved for local development tools while ensuring the stability of the production system.

Scale presents one of the most significant challenges when it comes to remote debugging. Developer observability tools address this issue by allowing users to associate any action (log, snapshot, or metric) with a tag. This is equivalent to applying the action across all agents (servers) linked to that tag.

For instance, a tag can be created based on the platform, such as Ubuntu 20, or based on the deployment type, like “Green.” This approach enables debugging at a massive scale, eliminating the need to predict which remote device will encounter the bug next.

By using tags, developer observability tools streamline debugging across multiple instances, making it more efficient and effective when working with complex and large-scale systems. This method offers a powerful solution for addressing the challenges associated with debugging distributed applications in modern production environments.

PII reduction and blocklists

The sandbox represents just one aspect of security in developer observability tools. As previously discussed, one of the major risks associated with deploying such tools arises from internal developers potentially mishandling data, either intentionally or inadvertently. Moreover, it is essential to comply with regulatory and legal requirements.

To address these challenges, two separate tools are used: PII reduction and blocklists.

Personal Identifiable Information (**PII**) refers to private user data that must be protected. For example, consider a user object that contains a cached credit card number. Logging the User object would inadvertently expose the user’s credit card number in logs ingested throughout the system.

Such exposure may violate regulations and laws, as well as result in penalties for the company. PII reduction prevents well-known patterns (such as credit card numbers, e-mail addresses, and so on) from appearing in logs, mitigating potential abuse as logs are more widely accessible than the database.

Blocklists enable the restriction of specific files or classes from action insertion. This means that developers can be prevented from adding snapshots, logs, or other actions to files that might pose risks. For instance, a file handling the login process should be blocked by default. By implementing these two features, the security of the deployment can be significantly strengthened.

Conclusion

Monitoring and observability are crucial components in maintaining the health and performance of modern software systems. Embracing a comprehensive approach that combines traditional monitoring practices with developer observability enables developers to gain valuable insights into their applications, both at the infrastructure and code levels. By integrating tools such as Prometheus, Grafana, Micrometer, Spring Actuator, and developer observability platforms, teams can effectively diagnose issues, optimize performance, and ensure the reliability of their applications.

Points to remember

- Monitoring and observability are essential for maintaining the health, performance, and reliability of software systems.
- Traditional monitoring practices should be combined with developer observability to gain insights into the issues at the source code level.
- Tools such as Prometheus, Grafana, Micrometer, and Spring Actuator play a vital role in providing insights and visibility into application performance.
- Developer observability platforms, including log injection and snapshots, enable debugging in production environments without compromising system integrity.
- Implementing comprehensive monitoring and observability practices contributes to a proactive development culture and the long-term success of software projects.
- An observable system is a system in which we can answer questions about production with factual responses.

Multiple choice questions

1. Which of the following tools is primarily used for collecting and storing metrics in a monitoring and observability setup?

a. Spring actuator

b. Grafana

c. Micrometer

d. Prometheus

2. Which tool is commonly used for visualizing metrics and creating dashboards in a monitoring and observability setup?

a. Spring actuator

b. Grafana

c. Micrometer

d. Prometheus

3. In the context of developer observability, what is the primary purpose of using blocklists?

a. To restrict specific files or classes from action insertion.

b. To filter out irrelevant information from logs.

c. To categorize logs based on their content.

d. To prevent unauthorized access to sensitive data. A single Web service

Answers

1. d

2. b

3. a

---

1 Shai Almog (author of this book) was the first employee at Lightrun and still holds company stock

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech

happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

acceptance test 177
adapter pattern 45, 46
Advanced Message Queuing Protocol (AMQP) 248
agent statefullness 188
annotation system change 69
Ant 3
Apache Kafka
  about 249
  key features 249
APIs
  about 92
  HttpClient 92
  Panama 93, 94
  scoped values 95, 96
  sequenced collection 96, 97
  serialization filtering 94, 95
  structured concurrency 94
Application Performance Management (APM) 279
arrays 21
Aspect Oriented Programming (AOP) 137-139
Atomicity 160

Atomicity, Consistency, Isolation, and Durability (ACID) 160
atomic package 127, 128
authentication 237-242
authorization 237-242
availability 244
AWS Lambda
  using 263-268

B

Bitbucket Pipelines 186
branch protection
  about 194
  benefits 195
  configuration 195
builder pattern 42-44
build types
  erasure 23
  generics 22-25
built-in types
  about 19, 20
  arrays 21

C

Checkpoint/Restore in Userspace (CRIU) 228
CircleCI 186
cloud-based
  about 187
  versus on premise 187
cloud ecosystem 272, 273
cloud-native 226, 227
Cloud Native Computing Foundation (CNCF) 211, 283
collections 130
command pattern 52-54
common design patterns

about 34
adapter pattern 45, 46
builder pattern 42-44
command pattern 52-54
façade pattern 47
factory pattern 40, 41
iterator pattern 55, 56
observer pattern 51, 52
proxy pattern 48-51
Singleton 35, 37-39
CompletableFuture 129, 130
concurrent collections
ConcurrentHashMap 131
ConcurrentLinkedDeque 131
ConcurrentLinkedQueue 131
ConcurrentSkipListMap 131
ConcurrentSkipListSet 131
concurrent garbage collector
versus parallel garbage collector 73
ConcurrentHashMap 131
ConcurrentLinkedDeque 131
ConcurrentLinkedQueue 131
ConcurrentSkipListMap 131
ConcurrentSkipListSet 131
Consistency 160, 244
Consul 236
container orchestration
about 210
Kubernetes (k8s) 211-215
Skaffold 215-219
containers 206, 207
Continuous Delivery (CD) 185
Continuous Integration (CI)

about 185
agent statefullness 188
cloud, versus on premise 187
Continuous Integration (CI) tools
about 186
Bitbucket Pipelines 186
CircleCI 186
factors 186, 187
GitLab CI/CD 186
Jenkins 186
Travis CI 186
Coordinated Restore at Checkpoint (CRaC) 228
custom information
exposing, via Spring Boot actuator 289-291

D

Data Transfer Object (DTO) 147
date and time APIs 70
deadlock 112
debugging 25
default methods 68
Dependency Injection (DI) 137-139
deprecation of finalization 83
developer observability
about 291, 292
logs, injecting 292, 293
non-breaking breakpoints 293, 294
snapshots 293, 294
Docker 207-210
Durability 160

E

encapsulation 8-14
Envoy 237

erasure 23
error handling 169-172
eventual consistency 242-246
exception 84, 85
Executors class
  about 114
  factory methods 114

F

façade pattern 47
factory pattern 40, 41
factory pattern types
  abstract factory 41
  factory class 40
  factory method 40
functional programming 58-60
functional test 177
Function as a Service (FaaS) 262

G

G1 Garbage Collector 75
Gang of Four (GoF) design pattern 31
Garbage Collectors (GCs) 72
generational garbage collection 73
generics 22-25
GitHub Actions
  about 188-193
  branch protection 194
GitLab CI/CD 186
GraalVM 97, 269-272
GraalVM properties
  Native Image 98
  Polygot 97
Gradle 3
Grafana

about 285
key features 285

H

HelloWorld
example 5-7
HttpClient 92

I

Identity Access and Management (IAM) 263
immutability 56-58
implementation inheritance 15
Infrastructure as Code (IaC) 220-234
inheritance
about 14-17
implementation inheritance 15
interface inheritance 15
integration test 176
Inversion of Control (IoC) 137-139
Isolation 160
Istio 236
iterator pattern 55, 56

J

Java
debugging 26
project, setting up 3, 4
Java 8
about 65
annotation system change 69
date and time APIs 70
default methods 68
Lambda expressions 66, 67
method parameters reflection 69
method references 67

streams 68
Java build tools
Ant 3
Gradle 3
Maven 3
Java Database Connectivity (JDBC) 161-164
Java JDK
installing 2, 3
Java Management Extensions (JMX) 289
Java Native Interfaces (JNI) 93
Java Persistence Architecture (JPA) 161-169
Java Specification Request (JSR) 70
Java Virtual Machine (JVM) 221, 284
JDK Enhancement Proposal (JEP) 76
Jenkins 186
JSON Web Token (JWT) 238
JUnit 178-180
Just in Time (JIT) 76

K

Kubernetes (k8s) 211-215

L

Lambda expressions 66, 67
Linkerd 236
linting 196-201
lock
about 115-122, 125
advantages 115
Long Term Support (LTS) 65, 266

M

Maven 3
message broker 247
message-oriented middleware 247
message queue pattern 250

messaging 247, 248
method parameters reflection 69
method references 67
microbenchmark suite 76-78
micrometer
  about 286, 287
  key features 286, 287
microservices
  characteristics 234
  keywords 233
  versus monoliths 232, 256
mocking 181
Mockito
  about 181, 182
  light in-memory database, using 183
  performance matters 182, 183
Model-View-Controller (MVC) 149
modular monolith
  about 256
  benefit 259
  modules 258
  Modulith 257, 258
Modules AKA Jigsaw (Java 9) 71, 72
monitoring 278-280
monoliths
  about 254, 255
  versus microservices 232, 256
multi-threading
  working 104, 105
multi-threading, approaches
  cooperative 104
  preemptive 104

mutual exclusion (mutex) 107-109

N

Native Image 98
newProxyInstance method
  array of interfaces 50
  ClassLoader 50
  invocationHandler instance 51
notify() 109-111

O

Object-Oriented Design (OOD) 33, 34
Object Oriented Programming (OOP)
  about 8
  need for 32, 33
Object Relational Mapper (ORM) 164
objects 15, 98, 99
observability pillars
  about 280
  logs 281
  metrics 280
  traces 281, 282
observable system 282, 283
observer pattern 51, 52
on premise
  versus cloud-based 187
OOP principals
  about 8
  encapsulation 8-14
  inheritance 14-17
  polymorphism 17-19
OpenJDK 2

P

Panama 93, 94

parallel collector 74
parallel garbage collector
  versus concurrent garbage collector 73
parallelism 106
Partition tolerance 244
pattern-matching instanceof 88
performance test 177
Personal Identifiable Information (PII) 294
PII blocklists 294, 295
PII reduction 294, 295
Platform Agnostic Security Token (PASETO) 238
Platform as a Service (PaaS) 263
point-to-point pattern 250
Polygot 97
polymorphism 17-19
primitives 98, 99
private methods
  adding, in interface 85
project Loom 80-82
Prometheus
  about 283, 284
  key features 283, 284
proxy pattern 48-51
publish-subscribe (pub-sub) pattern
  about 250-254
  characteristics 250
  usage 251

Q

queues 130

R

RabbitMQ
  about 248

key features 248
race condition 113
Read Eval Print Loop (REPL) 16
record patterns
defining 90
records 90
REpresentational State Transfer (REST) 146
REST API 146-149
REST principles
cacheability 146
client-server 146
code on demand 147
layered system 147
statelessness 146
uniform interface 147
Resume Driven Design (RDD) 211
Role Based Access Control (RBAC) 239

S

Saga pattern 243
scoped values 95
sealed classes 87, 88
Semaphore class 125
sequenced collection 96, 97
serial collector 74
serialization filtering 94, 95
serverless
about 262, 263
key aspects 262
service mesh
about 234-237
core principles 235
proxy 235, 236
Shenandoah GC 75

shorter-fluent syntax 66
Singleton 35-39
Skaffold
  about 215-219
  features 215, 216
Spring 136
Spring Boot 140-146
Spring Boot actuator
  about 287
  configuring 288, 289
  custom information, exposing via 289-291
  enabling 288, 289
  key features 287, 288
Spring Cloud Stream
  about 249
  key features 250
Spring Initializr
  about 140
  reference link 141
Spring MVC 149, 150
Spring Native
  about 221, 222
  implementing 223-226
  options 227, 228
  problems 222, 223
stateful agents 188
stateless agents 188
streams 68
string templates 90, 91
structured concurrency 94
Structured Query Language (SQL)
  about 159
  basics 159

switch expression 85-87
synchronizers 125-127

T

Terraform
  reference link 220
Test Driven Development (TDD)
  about 183, 184
  problem 184, 185
testing theory 176
test types
  acceptance test 177
  functional test 177
  integration test 176
  performance test 177
  unit test 176
text blocks 89, 90
thread pools
  about 113
  Executors class 114
thread safety 106, 107
throughput collector 74
Thymeleaf
  about 149, 152-158
  benefits 151
Travis CI 186

U

Unified Modeling Language (UML) 33
unit test 176
Universal Unique Identifier (UUID) 166
unnamed patterns 91, 92
UTF-8 by default 83

V

variables 91, 92
var keyword 85
Virtual Machine (VM)
  about 71
  generational garbage collection 73
  Modules AKA Jigsaw (Java 9) 71, 72
  serial collector 74
  Shenandoah (Java 12) 72
virtual threads
  about 78, 79
  project Loom 80-82

W

wait() 109-111

Z

Z Garbage Collector (ZGC) 75